AFRL-IF-WP-TR-2000-1507

THE STANFORD IBM MANAGEMENT OF MULTIPLE INFORMATION SYSTEMS (TSIMMIS) -- STANFORD

HECTOR GARCIA-MOINA, PH.D.

STANFORD UNIVERSITY
OFFICE OF SPONSORED PROGRAM
125 PANAMA
STANFORD, CA 94305-4125

MARCH 2000

FINAL REPORT FOR 09/30/1993 – 09/30/1999

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE OH 45433-7334

NOTICE

Using Government drawings, specifications, or other data included in this document are for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell patented invention that may relate to them.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

Charles P. Satterthwaite

CHARLES P. SATTERTHWAITE, Project Engineer
Embedded Information System Engineering Branch
AFRL/IFTA

James S. Williamson

JAMES S. WILLIAMSON, Chief
Embedded Information System Engineering Branch
AFRL/IFTA

Eugene C Blackburn

EUGENE C. BLACKBURN, Chief
Information Technology Division
AFRL/IFT

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MARCH 2000	FINAL REPORT FOR 09/30/1993 - 09/30/1999

4. TITLE AND SUBTITLE
THE STANFORD IBM MANAGEMENT OF MULTIPLE INFORMATION SYSTEMS (TSIMMIS) -- STANFORD

5. FUNDING NUMBERS
C F33615-93-1-1339
PE 62301
PR A004
TA 01
WU 01

6. AUTHOR(S)
HECTOR GARCIA-MOINA, PH.D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
STANFORD UNIVERSITY
OFFICE OF SPONSORED PROGRAM
125 PANAMA
STANFORD, CA 94305-4125

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AFB, OH 45433-7334
POC: CHARLES P. SATTERTHWAITE, AFRL/IFTA, 937-255-6548 EXT. 3584

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-WP-TR-2000-1507

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

OBJECTIVE: to improve the access and utility of heterogeneously distributed information.
BACKGROUND: DARPA awarded this effort as sister effort to TSIMMIS-IBM. This work develops techniques to access, integrate, and utilize distributed heterogeneous information.

14. SUBJECT TERMS
Heterogeneous distributed information, Intelligent Query Processing, Databases

15. NUMBER OF PAGES
121

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

The TSIMMIS Project
Final Report

The goal of the Tsimmis Project was to develop tools that facilitate the rapid integration of heterogeneous information sources that may include both structured and unstructured data. The project developed components that extract properties from unstructured objects, that translate information into a common object model, that combine information from several sources, that allow browsing of information, and that manage constraints across heterogeneous sites. Tsimmis was a joint project between Stanford and the IBM Almaden Research Center.

A common problem facing many organizations today is that of multiple, disparate information sources and repositories, including databases, object stores, knowledge bases, file systems, digital libraries, information retrieval systems, and electronic mail systems. Decision makers often need information from multiple sources, but are unable to get and fuse the required information in a timely fashion due to the difficulties of accessing the different systems, and due to the fact that the information obtained can be inconsistent and contradictory. The TSIMMIS tools support integrated access to multiple information sources, and ensure that the information obtained is consistent.

The enclosed papers give a comprehensive summary of the project and describe the major results obtained. In short, the results include:

- Wrapper building technology for accessing heterogeneous sources;
- Extraction technology for converting HTML web information to more structured form;
- Mediation technology for combining information from different sources; and
- Constrain management technology for coordinating multiple sources

The TSIMMIS Project: Integration of Heterogeneous Information Sources*

Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer,
Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

last-name@cs.stanford.edu

Abstract

The goal of the Tsimmis Project is to develop tools that facilitate the rapid integration of heterogeneous information sources that may include both structured and unstructured data. This paper gives an overview of the project, describing components that extract properties from unstructured objects, that translate information into a common object model, that combine information from several sources, that allow browsing of information, and that manage constraints across heterogeneous sites. Tsimmis is a joint project between Stanford and the IBM Almaden Research Center.

1 Overview

A common problem facing many organizations today is that of multiple, disparate information sources and repositories, including databases, object stores, knowledge bases, file systems, digital libraries, information retrieval systems, and electronic mail systems. Decision makers often need information from multiple sources, but are unable to get and fuse the required information in a timely fashion due to the difficulties of accessing the different systems, and due to the fact that the information obtained can be inconsistent and contradictory.

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Reid and Polly Anderson Faculty Scholar Fund, the Center for Integrated Systems at Stanford University, and by Equipment Grants from Digital Equipment Corporation and IBM Corporation.

The goal of the TSIMMIS[1] project is to provide tools for accessing, in an integrated fashion, multiple information sources, and to ensure that the information obtained is consistent. Numerous other recent projects have similar goals, of course. Before describing the differences between Tsimmis and other data integration projects, let us give an overview of the Tsimmis architecture, describing the functions of the various components and the philosophy of our approach. Refer to Figure 1.

1.1 Translators and Common Model

Figure 1 shows a collection of (disk-shaped) heterogeneous information sources. Above each source is a *translator* (or *wrapper*) that logically converts the underlying data objects to a common information model. To do this logical translation, the translator converts queries over information in the common model into requests that the source can execute, and it converts the data returned by the source into the common model.

For the Tsimmis project we have adopted a simple *self-describing* (or *tagged*) object model. Similar models have been in use for years; we call our version the *Object Exchange Model*, or *OEM*. OEM allows simple nesting of objects, and a complete specification is given in Section 2. The fundamental idea is that all objects, and their subobjects, have *labels* that describe their meaning. For example, the following object represents a Fahrenheit temperature of 80 degrees:

⟨temp-in-Fahrenheit, int, 80⟩

where the string "temp-in-Fahrenheit" is a human-readable label, "int" indicates an integer value, and "80" is the value itself. If we wish to represent a complex object, then each component of the object has its own label. For example, an object representing a set of two temperatures may look like:

[1] As an acronym, TSIMMIS stands for "The Stanford-IBM Manager of Multiple Information Sources." In addition, Tsimmis is a Yiddish word for a stew with "heterogeneous" fruits and vegetables integrated into a surprisingly tasty whole.

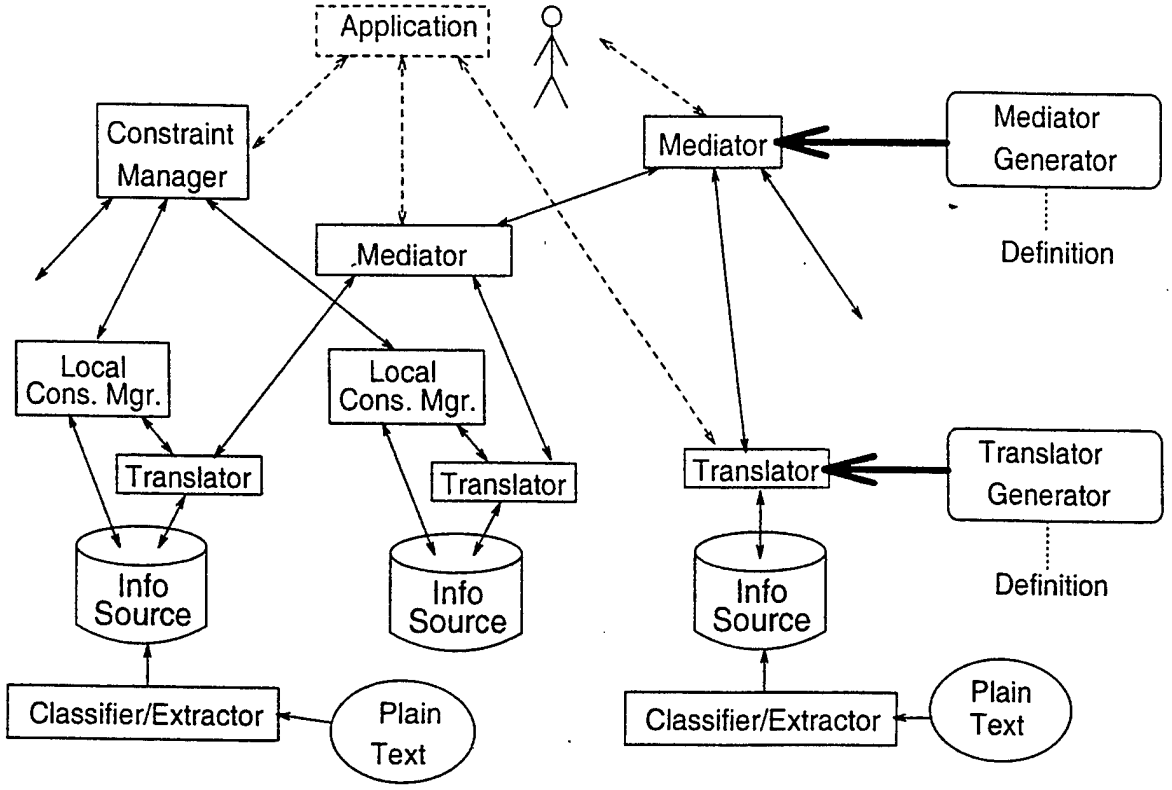


Figure 1: Tsimmis Architecture

⟨set-of-temps, set, $\{cmp_1, cmp_2\}$⟩
cmp_1: ⟨temp-in-Fahrenheit, int, 80⟩
cmp_2: ⟨temp-in-Celsius, int, 20⟩

OEM is very simple, while providing the expressive power and flexibility needed for integrating information from disparate sources. We also have developed a query language, *OEM-QL*, for requesting OEM objects. OEM-QL is an SQL-like language extended to deal with labels and object nesting; see Section 2.

1.2 Mediators

Above the translators in Figure 1 lie the *mediators*. A mediator is a system that refines in some way information from one or more sources [31]. A mediator embeds the knowledge that is necessary for processing a specific type of information. For example, a mediator for "current events" might know that relevant information sources are the AP Newswire and the New York Times database. When the mediator receives a query, say for articles on "Bosnia," it will know to forward the query to those sources. The mediator may also process answers before forwarding them to the user, say by converting dates to a common format, or by eliminating articles that duplicate information. While the task of converting dates is probably straightforward, the task of eliminating duplicate information could be very complex,—figuring out that two articles written by different authors say "the same thing" requires real intelligence. In Tsimmis we are focusing on relatively simple mediators based on patterns or rules. Still, even simple mediators can perform very useful information processing and merging tasks.

Implementing a mediator can be complicated and time-consuming, but we believe that much of the coding involved in mediators can be automated. Hence, one important goal of the Tsimmis project is to automatically or semi-automatically generate mediators from high level descriptions of the information processing they need to do. This is illustrated by the *mediator generator* box on the right side of Figure 1. Similarly, we provide a *translator generator* that can generate OEM translators based on a description of the conversions that need to take place for queries received and results returned. This component, also illustrated in Figure 1, significantly facilitates the task of implementing a new translator.

1.3 System and User Interfaces

Mediators export an interface to their clients that is identical to that of translators. Both translators and mediators take as input OEM-QL queries and return OEM objects. Hence, end users and mediators can

obtain their information either from translators and/or other mediators. This approach allows new sources to become useful as soon as a translator is supplied, it allows mediators to access new sources transparently, and it allows mediators to be "stacked," performing more and more processing and refinement of the relevant information.

End users (top of Figure 1) can access information either by writing applications that request OEM objects, or by using one of the generic browsing tools we have developed. Our most recent browsing tool provides access through *Mosaic* or other *World-Wide-Web* viewers [4,29]. The user writes a query on an interactive world-wide-web page, or selects a query from a menu. The answer is received as a hypertext document. The root of this document shows one or more levels of the answer object, with hypertext links available to take the user to portions of the answer that did not appear on the root document. This tool provides a mechanism for exploring heterogeneous information sources that is easy to interact with and that is based on a commonly used interface. The browser is described in more detail in Section 3.

1.4 Labels and Mediator Processing

It is important to note that there is no global database schema, and that mediators can work independently. For instance, to build a mediator it is only necessary to understand the sources that the mediator will use. In fact, it is not even necessary to fully understand the sources used. For example, returning to our "current events" mediator, suppose one source exports objects with subobjects labeled by *title*, *date*, *author*, and *country*. The mediator might always pass the *author* and *country* subobjects to its client with no additional processing. Now suppose a second source provides *topic* and *date* subobjects. The mediator might convert the dates from both sources into a common format, and it will know how to convert a mediator query about the subject of an article into the appropriate *topic* or *title* queries to be sent to the sources.

When a mediator simply passes subobjects to its clients (as in *author* and *country* above), it might append the source name to the labels so that the client can interpret the objects correctly. For example, a mediator subobject might have label *NYTimes.author*, indicating that this author is from the New York Times source and follows its conventions for authors. Another object might have the label *AP.author*. (Of course, the mediator could also make the formats consistent and export subobjects with label *author*, but here we are illustrating a simple mediator that does not do such processing.)

The key points are that a mediator does not need to understand all of the data it handles, and no person or software component needs to have a global view of all the information handled by the system.

1.5 Constraint Management

Another important component in the Tsimmis architecture is constraint management, illustrated in Figure 1 by a *Constraint Manager* and two *Local Constraint Managers*. *Integrity constraints* specify semantic consistency requirements over stored information; such constraints arise even when the information resides in loosely coupled, heterogeneous systems. For example, a construction company keeps data about a building under construction. This data must be consistent with the architect's design (e.g., walls must be in the same places), which may be stored in an entirely different system. Constraint management in the distributed, heterogeneous environments addressed by Tsimmis is a much more difficult and complex problem than constraint management in centralized systems: Transactions across multiple information sources usually are not provided, and each information source may support different capabilities for accessing and monitoring the data involved in a constraint.

In current environments, constraints across heterogeneous information sources usually are monitored or enforced by humans, in an ad-hoc fashion (or, frequently, not checked at all). For example, an architect may freeze the building design and send the latest specifications to the construction company so that consistency is "guaranteed." Of course, it is clear that these ad-hoc mechanisms do not work well in general; in our example, it is likely that the building may eventually not meet its specifications.

Since in a loosely coupled environment it is generally not possible to guarantee that every user or application sees consistent data every time it interacts with the system, the Tsimmis constraint manager enforces constraints with weaker guarantees than what a centralized system may provide. Tsimmis makes "relaxed" guarantees, e.g., a constraint is true from 8am to 5pm every day, or a constraint is true if some "Flag" is set. Ensuring relaxed consistency is especially challenging because one now has to deal with the timing of actions and of guarantees. However, the advantages of being able to handle relaxed guarantees in heterogeneous systems are significant; knowing precisely what holds and what does not hold, and when, will clearly lead to more trustworthy systems.

The Tsimmis constraint manager supports the definition of the *interfaces* that a source supports for the information involved in a constraint (e.g., can a trigger be set on a data item?), specification of the desired constraint (e.g., two items should have the same value),

and specification of the *strategy* that is to be followed for enforcing the constraint or for detecting violations. The Local Constraint Managers in Figure 1 are responsible for describing and supporting interfaces, while the Constraint Manager processes constraints and executes strategies. Note that the Constraint Manager actually is not centralized as illustrated in Figure 1, but rather is a set of distributed components that jointly manage constraints. Constraint management is described in more detail in Section 4.

1.6 Classification and Extraction

The final component of the Tsimmis architecture consists of the *Classifier/Extractors* shown at the bottom of Figure 1. Many important information sources are completely unstructured, consisting of plain files or incoming bit strings (e.g., from a newswire). Often it is possible to automatically classify the objects in such sources (e.g., is the file an email message, a text file, or a gif image?), and to extract key properties (e.g., creation date, author). The Classifier/Extractor performs this task, based on identifying simple patterns in the objects. The information collected by the Classifier/Extractor can then be exported (via a translator, if necessary) to the rest of the Tsimmis system, together with the raw data. The Classifier/Extractor component is based on the *Rufus* system developed at the IBM Almaden Research Center [25] and is not discussed further in this paper.

1.7 Related Work

There are a number of differences between integration of information sources in the Tsimmis project and other database integration efforts (e.g. [2,13,18,28] and many others):

- Tsimmis focuses on providing integrated access to very diverse and dynamic information. The information may be unstructured or semi-structured, often having no regular schema to describe it. The components of objects may vary in unpredictable ways (e.g., some pictures may be color, others black and white, others missing, some with captions and some without). Furthermore, the available sources, their contents, and the meaning of their contents may change frequently.

- Tsimmis assumes that information access and integration are intertwined. In a traditional integration scenario, there are two phases: an integration phase where data models and schemas (or parts thereof) are merged, and an access phase where data is fetched. In our environment, it may not be clear how information is merged until samples are viewed, and the integration strategy may change if certain unexpected data is encountered.

- Integration in our environment requires more human participation. In the extreme case, integration is performed manually by the end user. For example, a stock broker may read a report saying that IBM has named a new CEO, then retrieve recent IBM stock prices from a database to deduce that stock prices will rise. In other cases, integration may be automated by a mediator, but only after a human studies samples of the data, determines the procedure to follow, and develops an appropriate specification for the mediator generator.

In summary, the Tsimmis goal is *not* to perform fully automated information integration that hides all diversity from the user, but rather to provide a framework and tools to assist humans (end users and/or humans programming integration software) in their information processing and integration activities.

Regarding the constraint management aspects, there has been substantial prior work on database constraints, focusing on centralized databases (e.g., [14]), tightly-coupled homogeneous distributed databases (e.g., [12, 26]), or loosely-coupled heterogeneous databases with special constraint enforcement capabilities (e.g., [8,24]). The multidatabase transaction approach weakens the traditional notion of correctness of schedules (e.g., [5, 10]), but this approach cannot handle a situation in which different databases support different capabilities. In its modeling of time, our work has some similarity to work in temporal databases [27] and temporal logic programming [1], although our approach is closer to the event-based specification language in RAPIDE [19].

1.8 Remainder of Paper

In the rest of this paper we provide additional details on some of the Tsimmis components. In Section 2 we describe the OEM object model and its query language. In Section 3 we present the Tsimmis/Mosaic object browser. In Section 4 we outline the main components of the constraint management toolkit. In Section 5 we conclude, describe the status of the Tsimmis prototype, and discuss future directions of our work.

2 Object Exchange

As described in Section 1.1, our Object Exchange Model (OEM) is used as the unifying object model for information processed by Tsimmis components. Note that information need not actually be stored using OEM, rather OEM is used for the processing of logical queries, and for providing results to the user.

Each object in OEM has the following structure:

Label	Type	Value	Object-ID

where the four fields are:

- **Label:** A variable-length character string describing what the object represents. For each label a translator or mediator exports, it should provide a "help" page that describes (to a human) the meaning and use of the label. These help pages can be very useful during exploration of information sources, and for deciding how to integrate information.
- **Type:** The data type of the object's value. Each type is either an *atom* (or *basic*) type (such as *integer*, *string*, *real number*, etc.), or the type *set* or *list*. The possible atom types are not fixed and may vary from information source to information source.
- **Value:** A variable-length value for the object.
- **Object-ID:** A unique[2] variable-length identifier for the object or Λ (for null). The use of this field is described below.

In denoting an object on paper, we often drop the Object-ID field, i.e. we write ⟨label,type,value⟩, as in the examples in Section 1.1.

Suppose an object representing an employee has label *employee* and a set value. The set consists of three subobjects, a *name*, an *office*, and a *photo*. All four objects are exported by an information source IS through a translator, and they are being examined by a client C. The only way C can retrieve the employee object is by posing a query that returns the object as an answer.

Assume for the moment that the employee object is fetched into C's memory along with its three subobjects. The value field of the employee object will be a set of *object references*, say $\{o_1, o_2, o_3\}$. Reference o_1 will be the memory location for the name subobject, o_2 for the office, and o_3 for the photo. Thus, on the client side, the retrieved object will look like:

⟨employee, set, $\{o_1, o_2, o_3\}$⟩
  o_1: location of ⟨name, str, "some name"⟩
  o_2: location of ⟨office, str, "some office"⟩
  o_3: location of ⟨photo, bitmap, "some bits"⟩

On the information source side, the employee object may map to a real object of the same structure, or it may be an "illusion" created by the translator from other information. If IS is an object database, and the employee object is stored as four objects with object identifiers id_0 (employee), id_1 (name), id_2 (office), and id_3 (photo), then the retrieved object on the client side would have id_0 in the Object-ID field for the employee object, id_1 in the Object-ID field for the name object, and so on. The non-null Object-ID fields tell client C that the objects it has correspond to identifiable objects at IS. Suppose instead that IS is a relational database, and that the employee "object" is actually a tuple. Then, the name, office, and photo objects (attributes of the tuple) will not have object identifiers, and their Object-ID fields at the client side will be Λ (null).

So far we have assumed that the client retrieves the employee object and all of its subobjects. However, for performance reasons, the translator may prefer not to copy all subobjects. For example, if the photo subobject is a large bitmap with a unique identifier, it may be preferable to retrieve the name and office subobjects in their entirety, but retrieve only a "placeholder" for the photo object. In this case, the value field for the employee object at the client will contain $\{o_1, o_2, id_3\}$. This indicates that the name and office subobjects can be found at memory locations o_1 and o_2, but the photo subobject must be explicitly retrieved using id_3.

Note that, regardless of the representation used in set and list values, the translator always gives the client the illusion of an object repository. Thus, we can think of our employee object as:

⟨employee, set, $\{cmp_1, cmp_2, cmp_3\}$⟩
  cmp_1: ⟨name, str, "some name"⟩
  cmp_2: ⟨office, str, "some office"⟩
  cmp_3: ⟨photo, bits, "some bits"⟩

where each cmp_i is some mnemonic identifier for the subobject. We use this generic notation for examples throughout the remainder of this section.

As mentioned in Section 1, self-describing models have been used in many systems, including file systems [30], Lotus Notes [20], by Teknekron Software Systems [21], and for electronic mail. In many of these systems, nesting of objects is not allowed, so OEM can be viewed as a generalization of these models. OEM is simpler than conventional object models, but it does support the two key features required by object models [6]: *object nesting* and *object identity*.

Our primary reason for choosing a very simple model is to facilitate integration. As pointed out in [3], simple data models have an advantage over complex models when used for integration, since the operations to transform and merge data will be correspondingly simpler. Meanwhile, a simple model can still be very powerful: advanced features can be "emulated" when they are necessary. For example, if we wish to model

[2] We assume that identifiers are unique for each information source. Uniqueness across information sources can be achieved by, e.g., prepending each object identifier with a unique ID for the information source.

an employee class with subclasses *active* and *retired*, we can add a subobject to each employee object with label *subclass* and value "active" or "retired." Of course this is not identical to having classes and subclasses, since OEM does not force objects to conform to the rules for a class. While some may view this as a weakness of OEM, we view it as an advantage, since it lets us cope with the heterogeneity we expect to find in real-world information sources.[3]

2.1 Query Language and Examples

To request OEM objects from an information source, a client issues queries in a language we refer to as *OEM-QL*. OEM-QL adapts existing SQL-like languages for object-oriented models (e.g., [15, 16, 17, 23]) to OEM. Here we will give two examples to illustrate the "flavor" of OEM-QL; additional details and examples can be found in [22].

For the examples, suppose that we are accessing a bibliographic information source called `Biblio` with the object structure shown in Figure 2. (Note that we are using mnemonic object references.) Although much of this object structure is regular—components have the same labels and types—there are some irregularities. For example, the call number format is different for each document shown, and the n^{th} document uses a different structure for author information.

Example 2.1 Our first example retrieves the topic of each document for which "Ullman" is one of the authors:

```
SELECT bib.doc.topic
FROM Biblio
WHERE bib.doc.authors.author-ln = "Ullman"
```

Intuitively, the query's `WHERE` clause finds all paths through the subobject structure with the sequence of labels [`bib`, `doc`, `authors`, `author-ln`] such that the object at the end of the path has value "Ullman." For each such path, the `SELECT` clause specifies that one component of the answer object is the object obtained by traversing the same path, except ending with label `topic` instead of labels [`authors`, `author-ln`]. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, {o_1, o_2}⟩
  o_1: ⟨topic, str, "Databases"⟩
  o_2: ⟨topic, str, "Algorithms"⟩ □

[3]Note that some proposed interchange standards, e.g. CORBA's Object Request Broker [11], tend to be significantly more complex than OEM. We expect that if such standards are adopted, OEM could be used to provide a simpler, more "client-friendly" front end. Other proposed standards, such as ODMG's Object Database Standard [7], are directed towards interoperability and portability of object-oriented database systems, rather than towards facilitating object exchange in highly heterogeneous environments.

⟨bib, set, {doc_1, doc_2, ..., doc_n}⟩
  doc_1: ⟨doc, set, {au_1, top_1, cn_1}⟩
    au_1: ⟨authors, set, {au_1^1}⟩
      au_1^1: ⟨author-ln, str, "Ullman"⟩
    top_1: ⟨topic, str, "Databases"⟩
    cn_1: ⟨local-call#, integer, 25⟩
  doc_2: ⟨doc, set, {au_2, top_2, cn_2}⟩
    au_2: ⟨authors, set, {au_2^1, au_2^2,au_2^3}⟩
      au_2^1: ⟨author-ln, str, "Aho"⟩
      au_2^2: ⟨author-ln, str, "Hopcroft"⟩
      au_2^3: ⟨author-ln, str, "Ullman"⟩
    top_2: ⟨topic, str, "Algorithms"⟩
    cn_2: ⟨dewey-decimal, str, "BR273"⟩
  ⋮
  doc_n: ⟨doc, set, {au_n, top_n, cn_n}⟩
    au_n: ⟨one-author, str, "Michael Crichton"⟩
    top_n: ⟨topic, str, "Dinosaurs"⟩
    cn_n: ⟨fiction-call#, int, 95⟩

Figure 2: Object structure for example queries

Example 2.2 Our next example illustrates how variables are used to specify different paths with the same label sequence. This query retrieves each document for which both "Aho" and "Hopcroft" are authors:

```
SELECT bib.doc
FROM Biblio
WHERE bib.doc.authors.author-ln(a1) = "Aho"
  AND bib.doc.authors.author-ln(a2) = "Hopcroft"
```

Here, the query's `WHERE` clause finds all paths through the subobject structure with the sequence of labels [`bib`, `doc`, `authors`], and with two distinct path completions with label `author-ln` and with values "Aho" and "Hopcroft" respectively. The answer object contains one *doc* component for each such path. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, {o}⟩
  o: ⟨doc, set, {au_2, top_2, cn_2}⟩
    au_2: ⟨authors, set, {au_2^1, au_2^2,au_2^3}⟩
      au_2^1: ⟨author-ln, str, "Aho"⟩
      au_2^2: ⟨author-ln, str, "Hopcroft"⟩
      au_2^3: ⟨author-ln, str, "Ullman"⟩
    top_2: ⟨topic, str, "Algorithms"⟩
    cn_2: ⟨dewey-decimal, str, "BR273"⟩ □

2.2 Implementation

We have argued that OEM and its query language are designed to facilitate integrated access to heterogeneous data sources. To support this claim we have used the OEM model and language to integrate a variety of bibliographic information sources, including a conventional

library retrieval system, a relational database holding structured bibliographic records, and a file system with unstructured bibliographic entries. Using our OEM-based system, these sources are accessible through the Tsimmis browser (Section 3), allowing evaluation of queries and object exploration.

As an example, consider one of our operational translators that accesses the Stanford University *Folio* System. Folio provides access to over 40 repositories, including a catalog of the holdings of Stanford's libraries, and several commercial sources such as INSPEC that contain entries for Computer Science and other published articles. Folio is the most difficult of our information sources, partly because the translator must emulate an interactive terminal. The translator initially must establish a connection with Folio, giving the necessary account and access information. When the translator receives an OEM-QL query to evaluate, it converts the query into Folio's Boolean retrieval language. Then it extracts the relevant information from the incoming screens and exports the information as an OEM answer object. The Folio translator is written in C and runs as a server process on Unix BSD4.3 systems. Translators for the other bibliographic sources have involved substantially less coding because the underlying sources (e.g., a relational database) are much easier to use.

We also have implemented mediators that fuse information from multiple bibliographic sources. For example, one mediator provides a simple "union" of the sources, making the information appear as if it all comes from one source. Another mediator performs a "join" of two sources, combining entries that refer to the same document into a single entry that contains all information on the document available from either source.

Finally, we also have implemented *OEM Support Libraries* to facilitate the creation of future translators, mediators, and end-user interfaces. These libraries contain procedures that implement the exchange of OEM objects between a server (either a translator or a mediator) and a client (either a mediator, an application, or an interactive end-user). The Support Libraries handle all TCP/IP communications, transmission of large objects, timeouts, and many other practical issues. A Unix BSD4.3 and a Windows version of the package have been implemented and demonstrated. The Support Libraries are described in [22].

3 Object Browsing

The goal of the object browsing component of Tsimmis is to provide a platform-independent tool for displaying and exploring the OEM objects that are returned as a result of OEM-QL queries. Due to the nested structure of OEM objects, it is necessary to provide mechanisms that let end users navigate easily through the answer space, much like they would navigate through a tree structure. We have implemented *MOBIE* (MOsaic Based Information Explorer), a graphical browsing tool based on Mosaic and the World-Wide-Web [4, 29] for submitting Tsimmis queries and exploring the results. MOBIE lets end users connect to mediators or translators and specify queries using OEM-QL. An important advantage of using Mosaic as the basis for our user interface is its widespread use and popularity. (Mosaic currently operates on Unix workstations, on Macintosh computers, and on many PC's.) Hence, ultimately anyone on the internet should be able to use Tsimmis and MOBIE to explore any information source on the net, provided there is an appropriate translator or mediator available for it.

We illustrate MOBIE's operation by walking through a particular interaction. The first step in accessing information through MOBIE is to select a translator or mediator (henceforth referred to as TM) and connect to it. Figure 3 shows of MOBIE's home page[4] with a list of currently available TMs. The user may select any of the TMs on the list, enter its name in the provided box, and click on the Connect button. (Information shown below the CONNECT button is used to "fine-tune" the communication between the source and the client, and can generally be left in its default configuration.)

After the connection is established, a *Query Request* page (not shown) is displayed and the system is ready to accept an OEM-QL query. In the current version of MOBIE, queries must be entered by hand, meaning that the user must fill in the boxes provided on the screen (one box for the SELECT clause, one for the FROM clause, and one for the WHERE clause). However, future extensions will include the ability to select parameterized "frequently asked queries" by clicking on menus.

If a submitted query is valid and successfully executed by the TM, the answer object is returned to MOBIE and displayed on a *Query Result* page. Except for very small objects, to see the complete result the user will move through the structure of the answer object using MOBIE's navigational capabilities. This is best understood by thinking of the answer object as a tree (or a graph, in the most general case), where the atom objects are the leaves, and the set objects are the internal nodes. Initially, only the root of the answer object and its immediate subobjects are displayed on the Query Result page (not shown). For our bibliographic data, the root is typically a set containing a set of documents (labelled doc). The user can move from the current level in the object structure to a lower level by clicking on the FETCH

[4] Mosaic displays information through a series of text screens or *pages*, the first of which is always called the *home page*.

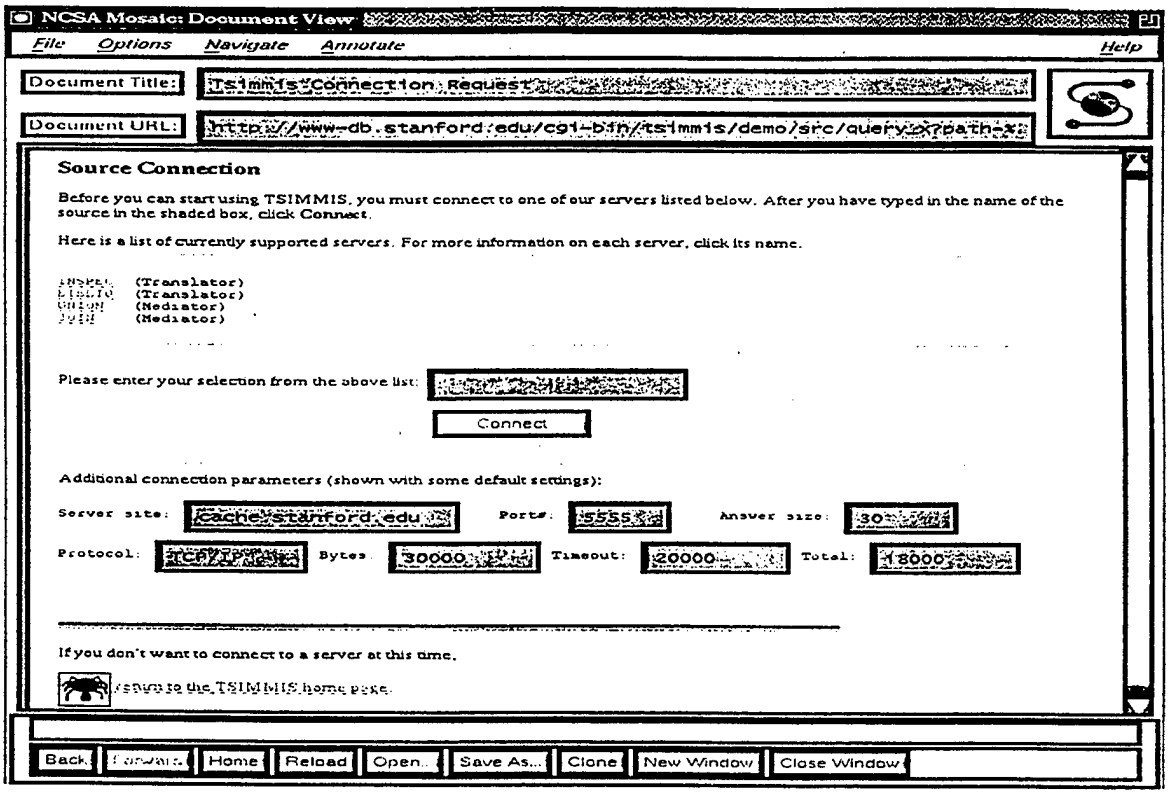


Figure 3: MOBIE's home page

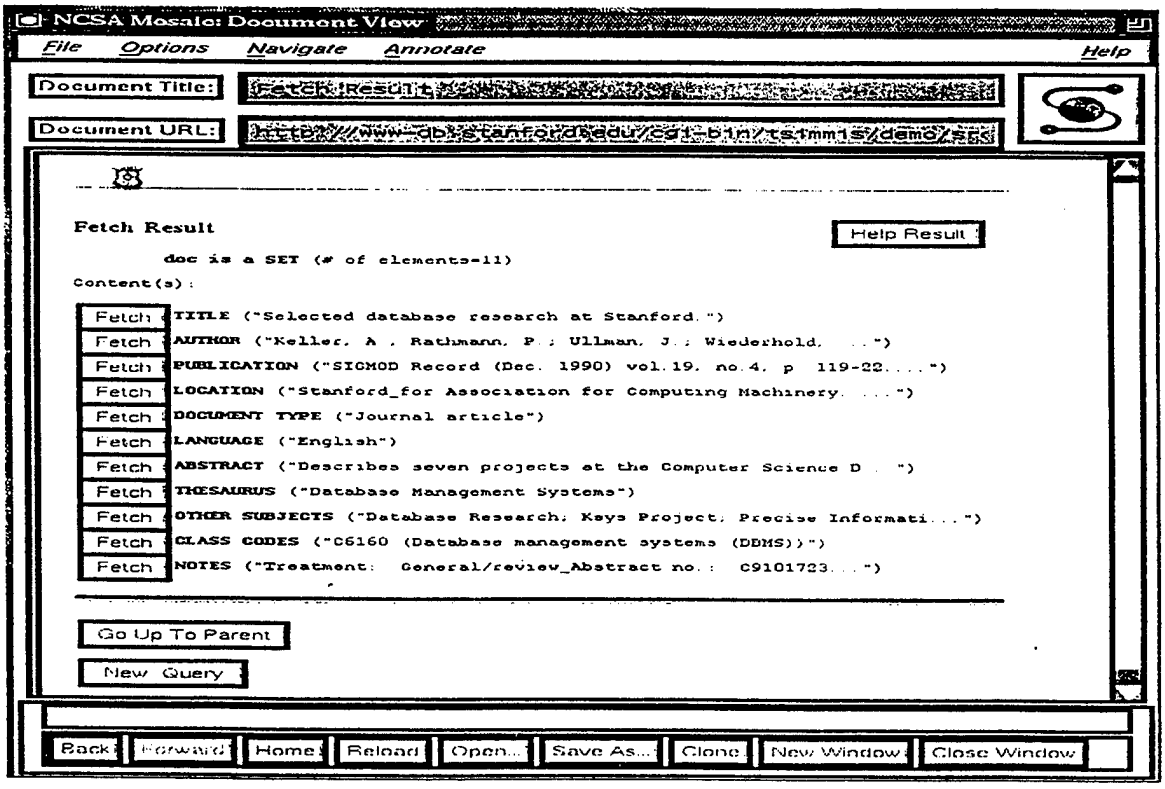


Figure 4: *Fetch Result* page displaying a selected document

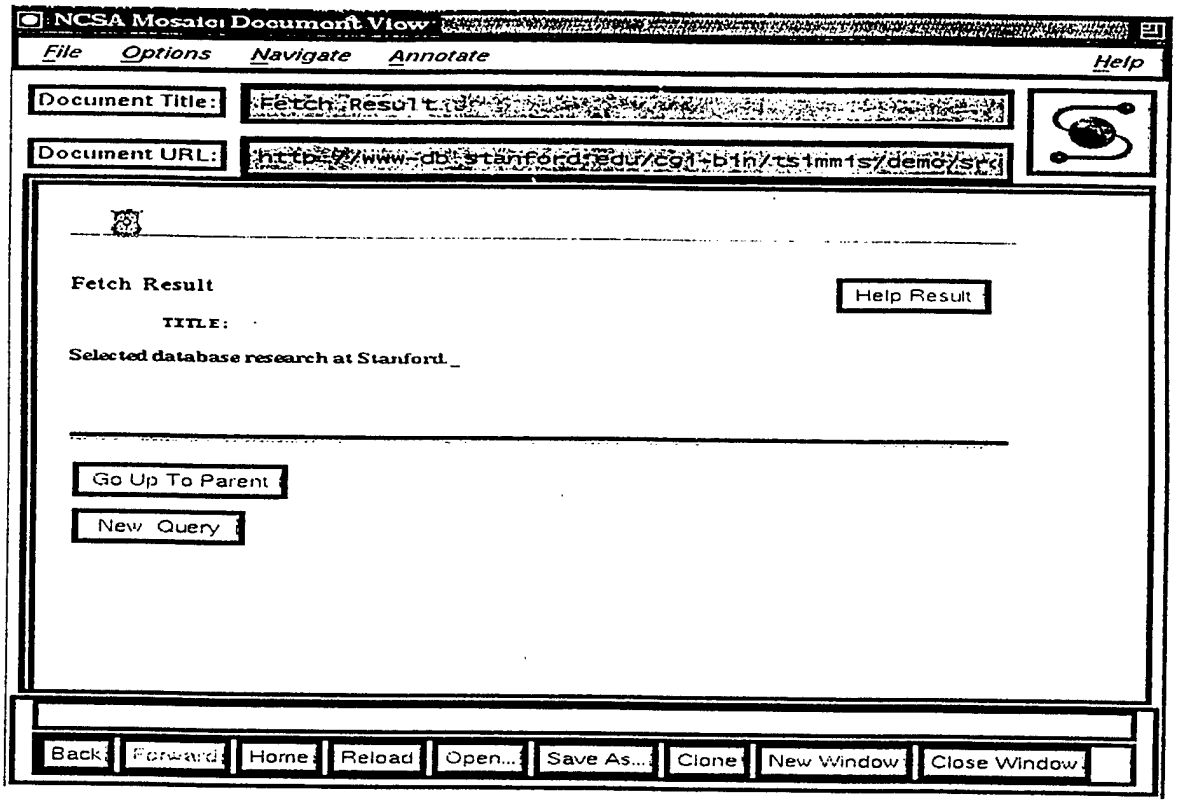


Figure 5: *Fetch Result* page showing the title of a document

buttons preceding each subobject. The result of clicking FETCH for one document in the intial query page is shown in Figure 4. The subobjects of this document are labelled Title, Author, etc., and their values can again be fetched by clicking on the FETCH buttons. For example, the result after clicking the TITLE button is shown in Figure 5. At this point we have a reached a leaf (or atom) in the answer space and cannot descend any further. The user can either backtrack to one of the parent objects by clicking on the Go Up To Parent button, or enter a new query by selecting New Query.

At any point in the session, the user can ask for help by selecting the Help Result button, which displays text on the meaning of a particular result object. As discussed in Section 2, each TM provides capabilities for describing (in English) the meaning of a label, and how to interpret the value of objects with that label. As an example, the help entry for the *author* label as returned by the translator would explain that author objects consist of a last name followed by a first name or initials. A MOBIE session is ended by selecting the Close Session button on the Query Request page.

4 Constraint Management

The Tsimmis Constraint Manager is based on a general formal framework we have developed for constraints in heterogeneous systems. Each information source (recall Figure 1) chooses an *interface* it can offer to the local constraint manager (LCM) for each of its data items involved in a multi-source constraint. The interface specifies how the data item may be read, written, and/or monitored by the LCM. Applications inform the constraint manager (CM) of constraints that need to be monitored or enforced. Based on the constraint and the interfaces available for the items involved in the constraint, the CM decides on the constraint management *strategy* it executes. This strategy monitors or enforces the constraint as well as possible using the interfaces offered by the information sources. The degree to which each constraint is monitored or enforced is formally specified by the *guarantee*. We briefly describe interfaces, strategies, and guarantees next. Complete formal specifications of each can be found in [9].

Interfaces are specified using a notation based on *events* and *rules*. As an example, we illustrate a simple "write interface" for a data item X. With this interface, the information source promises to write any requested value to X within five seconds. The interface is expressed as the rule:

$$WR(X, b) \rightarrow W(X, b);\ B \leq 5.$$

Here, $WR(X, b)$ represents a *write-request* event requesting operation $X := b$. The rule says that whenever such a write-request event occurs, a *write* event, $W(X, b)$, occurs within 5 time units. We assume that the interfaces for the data items involved in constraints are specified by a "constraint administrator"[5] at each site, based on the level of access and performance that can be provided to the CM for the data item. Currently, we rely on the users of our framework to verify that the interfaces specified do faithfully represent the actual systems.

The strategy for a constraint describes the algorithm used by the CM to monitor or enforce the constraint. Like interfaces, strategies are specified using a notation based on events and rules. In addition to performing operations on the data items involved in a constraint, strategies may evaluate predicates over the values of data items (obtained through read operations) and over private data maintained by the CM. As a simple example, consider the strategy description below, which issues a write request to Y within 7 seconds whenever a *notify* event is received from X. (A notify event represents the source notifying the CM of a write to a data item. Thus, e.g., $N(X, 5)$ represents the notification that a write $X := 5$ occurred.) This strategy might be used to maintain the constraint $X = Y$.

$$N(X, b) \rightarrow WR(Y, b);\ B \leq 7.$$

Rule-based strategy specifications are implemented using the host language of the CM. The translation from rules to host language is usually straightforward, and it may be achieved using a rule processing engine.

The guarantee for a constraint specifies the level of global consistency that can be ensured by the CM when a certain strategy for that constraint is implemented. Typically a guarantee is *conditional*, e.g., a guarantee might state that if no updates have been performed recently then the constraint holds, or if the value of a CM data item is true then the constraint holds. Guarantees are specified using predicates over values of data items and occurrences of certain events. For example, consider the following guarantee for a constraint $X = Y$:

$$(\text{Flag} = \text{true})@t \Rightarrow (X = Y)@@[t - \alpha, t - \beta].$$

This guarantee states that if the Boolean data item Flag (maintained by the CM) is true at time t, then $X = Y$ holds at all times during the interval $[t - \alpha, t - \beta]$. Note that this guarantee is weaker than a guarantee that $X = Y$ always holds, which is a very difficult guarantee to make in the heterogeneous, autonomous environments we are considering.

[5] The constraint administrator is an individual who is familiar with the structure and behavior of a given information source, much like a database administrator.

4.1 A Constraint Management Toolkit

As part of the Tsimmis prototype, we have built a toolkit that permits constraint management across heterogeneous and autonomous information systems. This toolkit allows us to enforce, for example, a copy constraint spanning data stored in a Sybase relational database and a file system, or an inequality constraint between a *whois*-like database and an object-oriented database.

Figure 6 depicts the architecture of our constraint management toolkit, which is based on the formal framework described above and interfaces with the Tsimmis architecture depicted in Figure 1. The Raw Information Sources (RIS) are what exist already at each site (for example, a relational database, a file system, or a news feed). The RISI is the source-specific interface offered by each RIS to its users and applications. For example, for a Sybase database, the RISI is based on a particular dialect of SQL, and includes details on how to connect to the server.

The CM-Translator is the module that implements the interfaces for each of its data items. The CM-Translator is specified by a configuration file called a CM-RID (for Raw Interface Description), which includes: (1) which interfaces (selected from a menu of interface types) are supported by the CM-Translator, and (2) how these interfaces are implemented using the underlying RISI.[6] The CM-Shells cooperate to execute the constraint management strategies. The CM-Shells are distributed rule engines that are configured by a Strategy Specification file.

We now describe how constraint administrators would use our toolkit to set up constraint management across multiple sources. The administrators at each site first decide on the CM-Interfaces they are willing to offer, selected from menu of predetermined interfaces provided by the toolkit. For example, if the underlying RIS provides triggers, then a notify interface may be offered; if not, perhaps a read/write interface can be offered. The choice also depends on the actions the administrator wants to allow. For instance, even if the RIS allows updates to the source, the administrator may disallow a write interface that lets the CM make changes to the local data. Each CM-RID file records the interfaces supported, as well as the specification of the RIS objects to which the interface applies.

[6] Note that the CM-Translator is responsible for translating between rule-based interface specifications (as described earlier) and source-specific operations. For translation of data and queries, a Tsimmis translator can be used. Hence, the CM-Translator together with the CM-RID comprise the Local Constraint Manager illustrated in Figure 1.

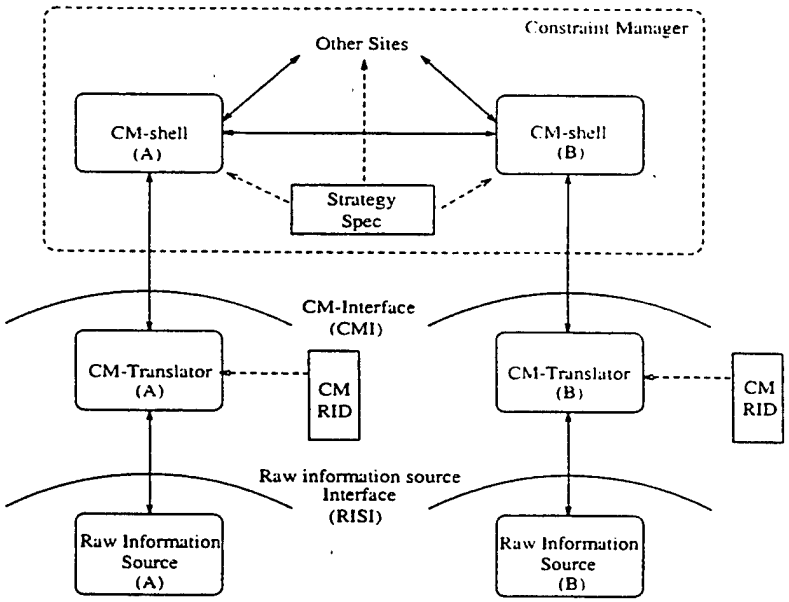


Figure 6: Constraint Management Toolkit Architecture

Next, the administrator uses a Strategy Design Tool (not shown in Figure 6) to develop the CM strategy. This tool takes as input the multi-source constraints; based on the available interfaces, it suggests strategies from its available repertoire. For each suggested strategy, the design tool can give the guarantee that would be offered. The result of this process is a Strategy Specification file, which is then used by the CM-Shells at run time. Note that knowledgeable administrators might choose to write their Strategy Specifications directly, bypassing the Design Tool.

5 Conclusion

In summary, the Tsimmis project is exploring technology for integrating heterogeneous information sources. Current efforts are focusing on translator and mediator generators, which should significantly reduce the effort required to access new sources and integrate information in different ways. We believe that the OEM model described here provides the right flexibility for handling unexpected heterogeneity.

Acknowledgements

We are grateful to Ed Chang and Jon Goldberg for their implementation efforts, to Ashish Gupta, Dallan Quass, and Anand Rajaraman for valuable comments, and to the entire Stanford Database Group for numerous fruitful discussions.

References

[1] Martin Abadi and Zohar Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–295, 1989.

[2] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

[3] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[4] T. J. Berners-Lee, R. Cailliau, and J.F. Groff. The world-wide web. *Computer Networks and ISDN Systems*, 25:454–459, 1992.

[5] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDBJ*, 1(2):181, October 1992.

[6] R. G. G. Cattell. *Object Data Management*. Addison-Wesley, 1991.

[7] R. G. G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[8] Stefano Ceri and Jennifer Widom. Managing semantic heterogeneity with production rules and persistent queues. In *VLDB*, pages 108–119, Dublin, Ireland, August 1993.

[9] Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom. Constraint management in loosely coupled
distributed databases. Technical report, Computer Science Department, Stanford University, 1993. Available through anonymous ftp from host `db.stanford.edu` as `pub/chawathe/1993/cm-loosely-coupled-dbs.ps`.

[10] Ahmed Elmagarmid, editor. *Special Issue on Unconventional Transaction Management,* Data Engineering Bulletin 14(1), March 1991.

[11] Object Request Broker Task Force. The Common Object Request Broker: Architecture and Specification, December 1993. Revision 1.2, Draft 29.

[12] Paul Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In *VLDB*, pages 581–591, Dublin, Ireland, August 1993.

[13] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

[14] M. Hammer and D.McLeod. A framework for database semantic integrity. In *Proceedings of the Second International Conference on Software Engineering*, pages 498–504, San Francisco, California, October 1976.

[15] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[16] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[17] H. F. Korth and M. A. Roth. Query languages for nested relational databases. In *Nested Relations and Complex Objects in Databases*, pages 190–204. Springer-Verlag, 1989.

[18] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[19] David C. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 1994.

[20] D. S. Marshak. Lotus Notes release 3. *Workgroup Computing Report*, 16:3–28, 1993.

[21] B. Oki et al. The information bus—an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[22] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings Data Engineering Conference*, Taipei, Taiwan, March 1995.

[23] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13:389–417, 1988.

[24] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *COMP*, 24(12):46–51, December 1991.

[25] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The rufus system: Information organization for semi-structured data. In *VLDB*, pages 97–107, Dublin, Ireland, August 1993.

[26] Eric Simon and Patrick Valduriez. Integrity control in distributed database systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 621–632, 1986.

[27] Richard Snodgrass. Temporal databases. *COMP*, 19(9):35–42, September 1986.

[28] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[29] Steven J. Vaughan-Nichols. How to glue together mosaic (internet browser) (tutorial). *Government Computer News*, 13(15):33, July 1994.

[30] G. Wiederhold. *File Organization for Database Design.* McGraw Hill, New York, 1987.

[31] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

Object Exchange Across Heterogeneous Information Sources*

Yannis Papakonstantinou
Hector Garcia-Molina
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
{yannis,hector,widom}@cs.stanford.edu

Abstract

We address the problem of providing integrated access to diverse and dynamic information sources. We explain how this problem differs from the traditional database integration problem and we focus on one aspect of the information integration problem, namely *information exchange*. We define an object-based information exchange model and a corresponding query language that we believe are well suited for integration of diverse information sources. We describe how the model and language have been used to integrate heterogeneous bibliographic information sources. We also describe two general-purpose libraries we have implemented for object exchange between clients and servers.

1 Introduction

A significant challenge facing the database field in recent years has been the integration of heterogeneous databases. Enterprises tend to represent their data using a variety of conflicting data models and schemas, while users want to access all data in an integrated and consistent fashion. There has been substantial progress on database integration techniques [1,9,13,19]; in addition, emerging standards such as SQL3 are aimed at eliminating many of the problems.

At the same time, however, the problem of integration has become much more challenging because users want integrated access to *information*—data stored not just in standardized SQL databases, but also in, e.g., object repositories, knowledge bases, file systems, and document retrieval systems. In addition, users want to integrate this information with "legacy" data, and even with data that is not stored but rather arrives on-line, e.g. over a news wire. As an example, consider a stock broker tracking a company, say IBM. The broker's information sources may include IBM product announcements, the stock market ticker tape, IBM profit/loss statements, news articles, structured databases containing historical information (dividends per year), personnel information (the 100 top-paid executives), general information (the Fortune 500), and so on. Queries may range

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Reid and Polly Anderson Faculty Scholar Fund, the Center for Integrated Systems at Stanford University, and by Equipment Grants from Digital Equipment Corporation and IBM Corporation.

from simple ones over a single source (e.g., *What were IBM sales in 1990?*), to simple ones involving multiple sources (e.g., *Get all recent news items where an IBM executive is mentioned*), to complex analyses (e.g., *Is IBM stock a good buy today?*).

Although there are many similarities, integrating a disparate set of information sources differs from the integration of conventional databases in the following ways:

- Many of the sources contain data that is unstructured or semi-structured, having no regular schema to describe the data. For example, a source may consist of free-form text; even if the text does have some structure, the "fields" (e.g., author, title, etc.) may vary in unpredictable ways.

- The environment is dynamic. The number of sources, their contents, and the meaning of their contents may change frequently. For example, the stock broker's company may add or drop an information source depending on its cost and usefulness; a source that predicts a company's earnings may periodically redefine how it computes the earnings.

- Information access and integration are intertwined. In a traditional environment, there are two phases: an integration phase where data models and schemas are combined, and an access phase where data is fetched. In our environment, it may not be clear how information is combined until samples are viewed, and the integration strategy may change if certain unexpected data is encountered.

- Integration in our environment requires more human participation. In the extreme case, integration is performed manually by the end user. For example, the stock broker may read a report saying that IBM has named a new CEO, then retrieve recent IBM stock prices from a database to deduce that stock prices will rise. In other cases, integration may be automated, but only after a human studies samples of the data and determines the procedure to follow. For example, a human may write a program that extracts yearly sales figures from IBM letters to stockholders and then "joins" this data (by year) with a table of dividends.

In light of these differences and difficulties, we believe that the goal is *not* to perform fully automated information integration that hides all diversity from the user, but rather to provide a framework and tools to assist humans (end users and/or humans programming integration software) in their information processing and integration activities. So, what should the framework and tools look like? There are at least three categories:

1. **Information exchange.** The various components of an information system need to exchange data objects (units of information), either for examination by an end user or for integration with other data objects. For this, there needs to be an agreement as to how objects will be requested, how they will be represented, what the semantic meaning of each object (and its components) is, and how objects are actually transported over a network. Once an exchange format is agreed upon, there need to be tools for translating between an information source and the exchange format.

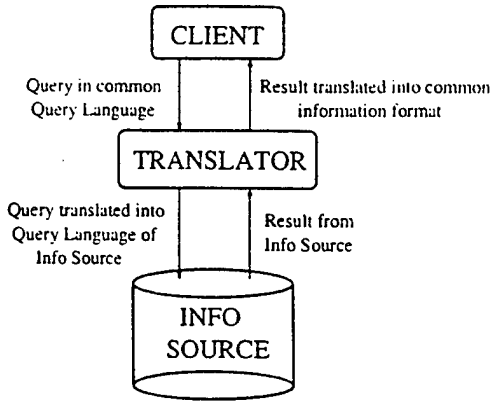


Figure 1: Communication through a translator

2. **Information discovery and browsing.** Users will want to explore the available information, discovering sources, browsing objects, and learning the semantics of objects and their components. Tools for information discovery and browsing allow humans (and ultimately software) to query for sources of interest, to request objects from sources, to navigate through objects (exploring their components), and to ask questions about the meaning of objects and their components.

3. **Mediators.** A mediator is a program that collects information from one or more sources, processes and combines it, and exports the resulting information [21]. For example, a mediator could be a program that collects IBM yearly stockholder reports, extracts key figures, and exports a table of yearly results. A second mediator might take this table and combine it with a stock price report to produce a trends analysis for IBM. We envision a variety of tools to assist the mediator writer, some resembling a programming environment, others presenting a menu of common ways of combining information.

In this paper we focus particularly on the information exchange problem discussed in point 1, since we believe this problem needs to be solved before browsing tools or mediators can be constructed. To motivate the information exchange problem further, consider an information source IS that contains bibliographic entries such as those found in many libraries. Some client C (human or otherwise) wishes to locate all books by author "J.D. Ullman" on the topic of "databases." Since IS and C are likely to be different, we need a common language and information format for communication. Client C uses the common language to express a query that requests the desired object. A front end to IS, which we call a *translator*, converts the query to a form that IS can process. When IS responds (with a set of bibliographic entries in some format), the translator converts the response into an object in the common format and transmits it to C. Finally, C may choose to translate the object (or the components it wants) into its own internal model. This form of communication is illustrated in Figure 1.

In Section 2 we present an "object exchange model" (OEM) that we believe is well suited for information exchange in heterogeneous, dynamic environments. OEM is flexible enough to

encompass all types of information, yet it is simple enough to facilitate integration; OEM also includes semantic information about objects. In Section 3 we describe the query language we have designed for requesting objects in OEM. In Section 4 we describe how we have used OEM to integrate several heterogeneous bibliographic information sources. In Section 5 we present a pair of general-purpose libraries we have implemented that support OEM object exchange between any client and server processes. The procedures in these libraries provide communication services, session handling, object memory management, and partial object fetches. Calls to these procedures are embedded in client programs. In Section 6 we conclude and discuss our ongoing work in information integration using OEM.

2 Object Exchange Model

The first question to be addressed is: with so many data models around, why do we need another one? In fact we do *not* need another new model. Rather, we adopt a model that has been in use for many years. The basic idea is very simple: each value we wish to exchange is given a *label* (or *tag*) that describes its meaning. For example, if we wish to exchange the temperature value 80 degrees Fahrenheit, we may describe it as:

⟨temperature-in-Fahrenheit, integer, 80⟩

where the string "temperature-in-Fahrenheit" is a human-readable label, "integer" indicates the type of the value, and "80" is the value itself. If we wish to exchange a complex object, then each component of the object has its own label. For example, an object representing a set of two temperatures may look like:

⟨set-of-temperatures, set, {$cmpnt_1$, $cmpnt_2$}⟩
    $cmpnt_1$ is ⟨temperature-in-Fahrenheit, integer, 80⟩
    $cmpnt_2$ is ⟨temperature-in-Celsius, integer, 20⟩

A main feature of OEM is that it is *self-describing*. We need not define in advance the structure of an object, and there is no notion of a fixed schema or object class. In a sense, each object contains its own schema. For example, "temperature-in-Fahrenheit" above plays the role of a column name, were this object to be stored in a relation, and "integer" would be the domain for that column.[1]

Note that, unlike in a database schema, a label here can play two roles: identifying an object (component), and identifying the meaning of an object (component). To illustrate, consider the following object:

⟨person-record, set, {$cmpnt_1$, $cmpnt_2$, $cmpnt_3$}⟩
    $cmpnt_1$ is ⟨person-name, string, "Fred"⟩

[1]Of course, if we are exchanging a set of objects where each object has the same structure and labels, then it would be redundant to transmit labels with every member of the set. We view this as a data compression issue and do not discuss it further here. From a logical point of view, we assume that each object in our model carries its own label.

$cmpnt_2$ is ⟨office-number-in-building-5, integer, 333⟩
$cmpnt_3$ is ⟨department, string, "toy"⟩

Like a column name in a relation, the label "person-name" identifies which component in the person's record contains the person's name. In addition, the label "person-name" identifies the meaning of the component—it is the name of a person. We would not expect to find a dog's name "Fido" or "Spot" in this component.

Thus, we suggest that labels should be as descriptive as possible. (For instance, in our example above, replacing "person-name" by "name" would not be advisable.) In addition, if an information source exports objects with a particular label, then we assume that the source can answer the question *What does this label mean?*. The answer should be a human-readable description—a type of "man page" (similar in flavor to Unix Manual pages). For example, if we ask the source that exports the above object about "person-name," it might reply with a text note explaining that this label refers to names of employees of a certain corporation, the names do not exceed 30 characters, and upper vs. lower case is not relevant.

It is particularly important to note that labels are relative to the source that exports them. That is, we do not expect labels to be drawn from an ontology shared by all information sources. For example, a client might see the label "person-name" originating from two different sources that provide personnel data for two different companies, and the label may mean something different for each source; the client is responsible for understanding the differences. If the client happens to be a mediator that exports combined personnel data for the two companies, then the mediator may choose to define a new label "generic-person-name" (along with a "man page"), to indicate that the information is not with respect to a particular company. Mediators are discussed further in Section 4.2.

We believe that a self-describing object exchange model provides the flexibility needed in a heterogeneous, dynamic environment. For example, personnel records could have fewer or more components than the ones suggested above; in our temperatures set, we could dynamically add temperatures in Kelvin, say. In spite of this flexibility, the model remains very simple.

As mentioned earlier, the idea of self-describing models is not new—such models have been used in a variety of systems (see Section 2.2 for a discussion of these models and systems). Consequently, the reader may at this point wonder why we are writing a paper about a self-describing model, if such models have been used for many years. A first reason is that we believe it is useful to formally cast a self-describing model in the context of information exchange in heterogeneous systems (something that has not been done before, to the best of our knowledge), and to extend the model to include object nesting as illustrated above. To do this, a number of issues had to be addressed, as will be seen in subsequent sections. A second reason is to provide an appropriate object request language based on the model. Our language is similar to nested-SQL languages; however, we believe that the use of labels within objects leads to a language that is more intuitive than nested-SQL (see Section 3).

2.1 Specification

Each object in OEM has the following structure:

Label	Type	Value	Object-ID

where the four fields are:

- **Label:** A variable-length character string describing what the object represents.
- **Type:** The data type of the object's value. Each type is either an *atom* (or *basic*) type (such as `integer`, `string`, `real number`, etc.), or the type `set` or `list`. The possible atom types are not fixed and may vary from information source to information source.
- **Value:** A variable-length value for the object.
- **Object-ID:** A unique variable-length identifier for the object or Λ (for `null`). The use of this field is described below.

In denoting an object on paper, we often drop the Object-ID field, i.e. we write ⟨label,type,value⟩, as in the examples above.

Object identifiers (henceforth referred to as OID's) may appear in set and list values as well as in the Object-ID field. We provide a simple example to show how sets (and similarly lists) are represented without OID's, and to motivate the kind of OID's that are used in OEM. Then we discuss OID's in set and list values.

Suppose an object representing an employee has label "employee" and a set value. The set consists of three subobjects, a "name," an "office," and a "photo." All four objects are exported by an information source IS through a translator, and they are being examined by a client C. The only way C can retrieve the employee object is by posing a query (see Section 3) that returns the object as an answer.

Assume for the moment that the employee object is fetched into C's memory along with its three subobjects. The value field of the employee object will be a set of *object references*, say $\{o_1, o_2, o_3\}$. Reference o_1 will be the memory location for the name subobject, o_2 for the office, and o_3 for the photo. Thus, on the client side, the retrieved object will look like:

⟨employee, set, {o_1, o_2, o_3}⟩
  o_1 is location of ⟨name, string, "some name"⟩
  o_2 is location of ⟨office, string, "some office"⟩
  o_3 is location of ⟨photo, bitmap, "some bits"⟩

On the information source side, the employee object may map to a real object of the same structure, or it may be an "illusion" created by the translator from other information. Suppose IS is an object database, and the employee object is stored as four objects with OID's id_0 (employee), id_1 (name), id_2 (office), and id_3 (photo). In this case, the retrieved object on the client side would

have id_0 in the Object-ID field for the employee object, id_1 in the Object-ID field for the name object, and so on. The non-null Object-ID fields tell client C that the objects it has correspond to identifiable objects at IS.

Now suppose instead that IS is a relational database, and that the employee "object" is actually a tuple. Hence, the name, office, and photo objects (attributes of the tuple) do not have OID's, so their Object-ID field at the client side will be Λ (null). The employee object may have an immutable tuple identifier, which can be used in the Object-ID field at the client. Alternatively, the employee's Object-ID field at the client might contain Λ, or it might contain an SQL statement that retrieves the employee record based on its key attribute.

So far we have assumed that the client retrieves the employee object and all of its subobjects. However, for performance reasons, the translator may prefer not to copy all subobjects. For example, if the photo subobject is a large bitmap, it may be preferable to retrieve the name and office subobjects in their entirety, but retrieve only a "placeholder" for the photo object. In this case, the value field for the employee object at the client will contain $\{o_1, o_2, id_3\}$. This indicates that the name and office subobjects can be found at memory locations o_1 and o_2, but the photo subobject must be explicitly retrieved using OID id_3.

Thus, at the client, sets and lists contain elements that may be of two forms, as follows. We assume there is an internal tag that indicates the form of each element.

- Local Object Reference: This identifies an object stored at the client. It will typically be a memory location, but if local objects are cached in an object database, then object references could be *Local OID's* in this database.

- Remote OID: This identifies an object at the information source.[2] Each Remote OID is either *lexical* or *non-lexical*. Lexical OID's are printable strings, and they may be specified directly in our query language (see Section 3). Non-lexical OID's are "black boxes," such as the tuple identifiers or SQL queries described above. Clients may pass non-lexical OID's to translators using special interfaces, but since the OID's are not printable, they cannot be used in queries. Remote OID's could be classified further by other properties [6], such as whether they are permanent or temporary [4]. (Or, OID's could include a "valid timestamp" specifying when they expire.) We do not consider these further classifications here, although we may incorporate these concepts in a future extension of our model.

Note that, regardless of the representation used in set and list values, the translator always gives the client the illusion of an object repository. Thus, we can think of our employee object as:

⟨employee, set, $\{cmpnt_1, cmpnt_2, cmpnt_3\}$⟩
  $cmpnt_1$ is ⟨name, string, "some name"⟩
  $cmpnt_2$ is ⟨office, string, "some office"⟩
  $cmpnt_3$ is ⟨photo, bits, "some bits"⟩

[2] We assume that identifiers are unique for each information source. Uniqueness across information sources can be achieved by, e.g., prepending each object identifier with a unique ID for the information source.

where each $cmpnt_i$ is some mnemonic identifier for the subobject. We use this generic notation for examples throughout the paper.

A final issue regarding OEM is that of duplicate objects at the client. Suppose, for example, that set object A at the information source has B and C as subobjects. Both B and C are of set type, and both have as subobjects the same object D. A query at a client retrieves A and all of its subobjects. Will the client have a single copy of object D, or will objects B and C point to different copies of D? Our model does *not* require a single copy of D at the client, since this would place a heavy burden on translators that are not dealing with real objects at the information source. However, if both copies of D have the same (non-null) Object-ID field, then the client can discern that the two objects correspond to the same object at the source. Also note that we do not require translators to discover cyclic objects at the source. Suppose, for example, that A has B as a subobject and B has A as a subobject. If the client fetches A from a "smart" translator, the translator would return only two objects, a copy of A and a copy of B. Each object's set value would be a reference for the other object. However, a "dumb" translator is free to return, say, four objects, A_1, B_1, A_2, B_2, where A_1 references B_1, B_1 references A_2, A_2 references B_2, and B_2 contains the empty set to indicate that for performance reasons the chain was not followed.

2.2 Related Models and Systems

In this section we contrast OEM with other similar models and systems. We focus particularly on the differences between OEM and more conventional object-oriented models, and we discuss the motivation behind our design of OEM.

Labeled fields are used as the basis of several data models or data formatting conventions. For example, a *tagged file system* [20] uses labels instead of positions to identify fields; this is useful when records may have a large number of possible fields, but most fields are empty. Electronic mail messages consist of label-value pairs (e.g. label "From" and value "yannis@cs.stanford.edu"). More recently, Lotus Notes [15] has used a label-value model to represent office documents, and Teknekron Software Systems [16] has used a self-describing object model for exchange of information in their stock trading systems. In [13] and [14] self-describing databases are proposed as a solution to obtaining the increased flexibility required by heterogeneous systems.

Recent projects on heterogeneous database systems (e.g., [1,3,11]) have applied object-oriented (OO) data models to the problem of database integration. OEM differs from these and other OO data models in several ways. First, OEM is an information *exchange* model. OEM does not specify how objects are stored at the source. OEM does specify how objects are received at a client, but after objects are received they can be stored in any way the client likes. OEM explicitly handles cross-system OID's (e.g., in Section 2.1 an employee object at the client points to a photo object at the source). In a conventional OO system there may also be client copies of server objects, but there the client copy is logically identical to the server copy and an application program at the client is not aware of the difference.

A very important difference between OEM and conventional OO models is that OEM is much simpler. OEM supports only *object nesting* and *object identity*; other features such as classes, methods, and inheritance are omitted. (Incidentally, [4] claims that the only two essential features of an OO data model are nesting and object identity.) Our primary reason for choosing a very simple model is to facilitate integration. As pointed out in [2], simple data models have an advantage over complex models when used for integration, since the operations to transform and merge data will be correspondingly simpler. Meanwhile, a simple model can still be very powerful: advanced features can be "emulated" when they are necessary. For example, if we wish to model an employee class with subclasses "active" and "retired," we can add a subobject to each employee object with label "subclass" and value "active" or "retired." Of course this is not identical to having classes and subclasses, since OEM does not force objects to conform to the rules for a class. While some may view this as a weakness of OEM, we view it as an advantage, since it lets us cope with the heterogeneity we expect to find in real-world information sources.[3]

The flexible nature of OEM can allow us to model complex features of a source in a simple way. For example, consider a deductive database that contains a parent relation and supports the recursive ancestor relation through derivation rules. If we wish to provide an OEM model of this data in which it is easy to locate a person's ancestors, we can make the object that corresponds to each person contain as subobjects the objects that correspond to his/her parents. It is then simple to pose a query in our OEM query language (see Section 3) that retrieves all of a person's ancestors. In addition, a user can browse through a person's "family tree" using the browsing facility described in Section 4.1.

A final distinct difference between OEM and conventional OO models is the use of labels in place of a schema. Clearly, it would be trivial to add labels to a conventional OO model (e.g., all objects could have an attribute called "label"). The only difference then is that in OEM labels are first-class citizens. We believe this small change makes interpretation and manipulation of objects more straightforward, as discussed in the next section. Note that the schema-less nature of OEM is particularly useful when a client does not know in advance the labels or structure of OEM objects. In traditional data models, a client must be aware of the schema in order to pose a query. In our model, a client can discover the structure of the information as queries are posed.

3 Query Language

To request OEM objects from an information source, a client issues queries in a language we refer to as *OEM-QL*. OEM-QL adapts existing SQL-like languages for object-oriented models to OEM.

The basic construct in OEM-QL is an SQL-like SELECT-FROM-WHERE expression. The syntax is:

[3]Note that some proposed interchange standards, e.g. CORBA's Object Request Broker [8], tend to be significantly more complex than OEM. We expect that if such standards are adopted, OEM could be used to provide a simpler, more "client-friendly" front end. Other proposed standards, such as ODMG's Object Database Standard [5], are directed towards interoperability and portability of object-oriented database systems, rather than towards facilitating object exchange in highly heterogeneous environments.

SELECT *Fetch-Expression*
FROM *Object*
WHERE *Condition*

The result of this query is itself an object, with the special label "answer":

$\langle$answer, set, $\{obj_1, obj_2, \ldots, obj_n\}\rangle$

Each returned subobject obj_i is a component of the object specified in the FROM clause of the query, where the component is located by the *Fetch-Expression* and satisfies the *Condition*. Details are given below. We assume that the *Object* in the FROM clause is specified using a lexical object-identifier, and that for every information source there is a distinguished object with lexical identifier "root." (Sources may or may not support additional lexical identifiers.) Certainly the query language may be extended with a call interface that allows non-lexical object identifiers in FROM clauses.

The *Fetch-Expression* in the SELECT clause and the *Condition* in the WHERE clause both use the notion of a *path*, which describes a traversal through an object using subobject structure and labels. For example, the path "bibliography.document.author" describes components that have label "author," and that are subobjects of an object with label "document" that is in turn a subobject of an object with label "bibliography." Paths are used in the *Fetch-Expression* to specify which components are returned in the answer object; paths are used in the *Condition* to qualify the fetched objects or other (related) components in the same object structure. A path specified in a *Fetch-Expression* may be terminated by the special symbol "OID," in which case only object identifiers are returned in the answer object, rather than the objects themselves.[4] A syntax and semantics for the basic constructs of OEM-QL is given in the Appendix. In the remainder of this section we provide a number of examples that serve to illustrate its capabilities.

For the examples, suppose that we are accessing a bibliographic information source with the object structure shown in Figure 2. (Note that we are using mnemonic object references; recall Section 2.) Let the entire object (i.e., the top-level object with label "bibliography") be the distinguished object with lexical object identifier "root". Note that although much of this object structure is regular—components have the same labels and types—there are some irregularities. For example, the call number format is different for each document shown, and the third document uses a different structure for author information.

Example 3.1 Our first example retrieves the topic of each document for which "Ullman" is one of the authors:

```
SELECT bibliography.document.topic
FROM root
WHERE bibliography.document.author-set.author-last-name = "Ullman"
```

[4] If there are qualifying objects without OID's, these objects are not returned in the answer object.

⟨bibliography, set, $\{doc_1, doc_2, \ldots, doc_n\}$⟩
doc_1 is ⟨document, set, $\{authors_1, topic_1, call\text{-}number_1\}$⟩
$authors_1$ is ⟨author-set, set, $\{author_1^1\}$⟩
$author_1^1$ is ⟨author-last-name, string, "Ullman"⟩
$topic_1$ is ⟨topic, string, "Databases"⟩
$call\text{-}number_1$ is ⟨internal-call-no, integer, 25⟩
doc_2 is ⟨document, set, $\{authors_2, topic_2, call\text{-}number_2\}$⟩
$authors_2$ is ⟨author-set, set, $\{author_2^1, author_2^2, author_2^3\}$⟩
$author_2^1$ is ⟨author-last-name, string, "Aho"⟩
$author_2^2$ is ⟨author-last-name, string, "Hopcroft"⟩
$author_2^3$ is ⟨author-last-name, string, "Ullman"⟩
$topic_2$ is ⟨topic, string, "Algorithms"⟩
$call\text{-}number_2$ is ⟨dewey-decimal, string, "BR273"⟩
⋮
doc_n is ⟨document, set, $\{authors_n, topic_n, call\text{-}number_n\}$⟩
$authors_n$ is ⟨single-author-full-name, string, "Michael Crichton⟩
$topic_n$ is ⟨topic, string, "Dinosaurs"⟩
$call\text{-}number_n$ is ⟨fiction-call-no, integer, 95⟩

Figure 2: Object structure for example queries

Intuitively, the query's `WHERE` clause finds all paths through the subobject structure with the sequence of labels [`bibliography`, `document`, `author-set`, `author-last-name`] such that the object at the end of the path has value "Ullman." For each such path, the `FROM` clause specifies that one component of the answer object is the object obtained by traversing the same path, except ending with label `topic` instead of labels [`author-set`, `author-last-name`]. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, $\{obj_1, obj_2\}$⟩
obj_1 is ⟨topic, string, "Databases"⟩
obj_2 is ⟨topic, string, "Algorithms"⟩ □

Example 3.2 Our second example illustrates the use of "wild-cards" and an existential `WHERE` clause. This query retrieves the topics of all documents with internal call numbers.

```
SELECT bibliography.?.topic
FROM root
WHERE bibliography.?.internal-call-no
```

The "?" label matches any label. Therefore, for this query, the `document` labels in Figure 2 could be replaced by any other strings and the query would produce the same result. By convention,

two occurrences of ? in the same query must match the same label unless variables are used (see below). Note that there is no comparison operator in the WHERE clause of this query, just a path. This means we only check that the object with the specified path exists; its value is irrelevant. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, $\{obj_1\}$⟩
    obj_1 is ⟨topic, string, "Databases"⟩ □

Example 3.3 In Example 3.2, the wild-card symbol ? was used to match any label. We also allow "wild-paths," specified by the symbol "*". Symbol * matches any path of length one or more.[5] Using *, the query in the previous example would be expressed as:

```
SELECT *.topic
FROM root
WHERE *.internal-call-no
```

The use of * followed by a single label is a convenient and common way to locate objects with a certain label in a complex structure. Similar to ?, two occurrences of * in the same query must match the same sequence of labels, unless variables are used. □

Example 3.4 Our next example illustrates how variables are used to specify different paths with the same label sequence. This query retrieves each document for which both "Aho" and "Hopcroft" are authors:

```
SELECT bibliography.document
FROM root
WHERE bibliography.document.author-set.author-last-name(a1) = "Aho"
  AND bibliography.document.author-set.author-last-name(a2) = "Hopcroft"
```

Here, the query's WHERE clause finds all paths through the subobject structure with the sequence of labels [bibliography, document, author-set], and with two distinct path completions with label author and with values "Aho" and "Hopcroft" respectively. The answer object contains one "document" component for each such path. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, $\{obj\}$⟩
    obj is ⟨document, set, $\{authors_2, topic_2, call\text{-}number_2\}$⟩
        $authors_2$ is ⟨author-set, set, $\{author_2^1, author_2^2, author_2^3\}$⟩
            $author_2^1$ is ⟨author-last-name, string, "Aho"⟩
            $author_2^2$ is ⟨author-last-name, string, "Hopcroft"⟩
            $author_2^3$ is ⟨author-last-name, string, "Ullman"⟩
        $topic_2$ is ⟨topic, string, "Algorithms"⟩
        $call\text{-}no_2$ is ⟨dewey-decimal, string, "BR273"⟩ □

[5] Note that our use of wild-card symbols is similar to, e.g., Unix, X-windows, etc.

Example 3.5 Our next example illustrates how object identifiers may be retrieved instead of objects.[6] This query retrieves the OID's for all documents with a Dewey Decimal call number:

```
SELECT *.OID
FROM root
WHERE *.dewey-decimal
```

In this query, since the path in the FROM clause ends with "OID," only object identifiers are returned. Hence, for the portion of the object structure shown in Figure 2 the query returns:

$\langle$answer, set, $\{id_1\}\rangle$

where id_1 is the OID for the object referred to as doc_2 in Figure 2. □

Example 3.6 Although we have used only equality predicates so far, OEM-QL permits any predicate to be used in the *Condition* of a WHERE clause. The predicates that can be evaluated for a given information source depend on the translator and the source. Suppose, for example, that a bibliographic information source supports a predicate called author that takes as parameters a document and the last name of an author; the predicate returns *true* iff the document has at least one author with the given last name. Then the query in Example 3.4 might be written as:

```
SELECT bibliography.document
FROM root
WHERE author(bibliography.document, "Aho")
  and author(bibliography.document, "Hopcroft")
```

One of the translators we have built (see Section 4) is for a bibliographic information source called *Folio* that does in fact support a rich set of predicates. All of the predicates supported by Folio are available to the client through OEM-QL. □

In the Appendix we provide a grammar for the basic OEM-QL syntax and a semantics specified as the answer object returned for an arbitrary query. The basic OEM-QL described in this paper is certainly amenable to extensions. For example, here we have allowed only one object in the FROM clause, so "joins" between objects cannot be described at the top level of a query. The language can easily be extended to allow multiple objects in the FROM clause. Similarly, the SELECT clause allows only one path to be specified; "constructors" can be added so that new object structures can be created as the result of a query. While these extensions are clearly useful, and we plan to incorporate them in the near future, we also expect that many translators (especially translators for unstructured and semi-structured information sources) will support only the basic OEM-QL (some may even support just a subset), since supporting the full extended language may result in unreasonable increase of the translator's complexity. One useful extension we plan for OEM-QL,

[6]Here the client is explicitly requesting OID's instead of objects. In other cases OID's may be retrieved instead of objects for efficiency; recall Section 2.1.

and we expect will be supported by most translators, is the ability to express queries about labels and object structure: we expect that clients will frequently need to "learn" about the objects exported by an information source before meaningful queries can be posed.

3.1 Related Languages

Many query languages for object-oriented and nested relational data models are based on an extension of SQL with path expressions, e.g. [10,11,12,17]. As stated earlier, OEM-QL can be viewed as an adaptation of these languages to the specifics of OEM.

In OEM-QL, path expressions range only over objects, while in most other languages they range over the schema and the objects. For example, consider the WHERE condition `document.author = "Smith"`. In OEM-QL, we simply find all objects with label document that have a subobject with label author and value "Smith." In a conventional OO language, we would have to identify a class document with an attribute named author. Then we would range over all objects of class document looking for the matching name. We believe that the simplicity of ranging over objects only leads to a more intuitive language and a more compact language definition.

A significant feature of OEM-QL is that it lets us query information sources where there is no regular schema. A conventional language breaks down in such a case, unless one defines an object class for every possible type of irregular object. (Note that such a schema would have to be modified each time a different object appeared.) Of course, if a particular information source does have a schema and a regular structure, the translator for that source should take advantage of the schema. For example, suppose all objects are stored in a relational database, and the translator receives the WHERE condition `document.author = "Smith"`. The translator could first check that there is a relation document with attribute author and, if so, could use an index to fetch the matching objects. Thus, the fact that the model and language do not require a schema does not mean that a schema cannot be used for query processing.

4 Implementation of Translators, Browsers, and Mediators

We have argued that OEM and its query language are designed to facilitate integrated access to heterogeneous data sources. To support this claim, in this section we describe how we have applied OEM to a particular scenario. The scenario consists of a variety of bibliographic information sources, including a conventional library retrieval system, a relational database holding structured bibliographic records, and a file system with unstructured bibliographic entries. Using our OEM-based system, these sources are accessible through a general-purpose user interface that allows evaluation of queries and object exploration.

Our first operational translator accesses the Stanford University *Folio* System. Folio provides access to over 40 repositories, including a catalog of the holdings of Stanford's libraries, and several commercial sources such as INSPEC that contain entries for Computer Science and other published articles. Folio is the most difficult of our information sources, partly because the translator must

emulate an interactive terminal. The translator initially must establish a connection with Folio, giving the necessary account and access information. When the translator receives an OEM-QL query to evaluate, it converts the query into Folio's Boolean retrieval language. Then it extracts the relevant information from the incoming screens and exports the information as an OEM answer object. The Folio translator is written in C and runs as a server process on Unix BSD4.3 systems. We have also implemented several simple mediators that refine the objects exported by the translator (see Section 4.2). Translators for the other bibliographic sources are nearly complete—they have involved substantially less coding because the underlying sources (e.g., a relational database) are much easier to use. Our translators and mediators are discussed further in Section 4.2.

We have also implemented *OEM Support Libraries* to facilitate the creation of future translators, mediators, and end-user interfaces. These libraries contain procedures that implement the exchange of OEM objects between a server (either a translator or a mediator) and a client (either a mediator, an application, or an interactive end-user). The Support Libraries handle all TCP/IP communications, transmission of large objects, timeouts, and many other practical issues. A Unix BSD4.3 and a Windows version of the package have been implemented and demonstrated. The Support Libraries are described in Section 5.

Finally, we have implemented a *Heterogeneous Information Browser* that lets a user submit queries and explore resulting objects.[7] The Browser is implemented in Visual C++ and runs under Windows. The next subsection describes the Browser in more detail. We believe the Browser illustrates the desirability of a simple model and language from the point of view of a user who may not be familiar with the underlying information.

4.1 The Heterogeneous Information Browser

The Heterogeneous Information Browser (HIB) provides a graphical user interface for submitting queries and exploring results. We illustrate its operation by walking through a particular interaction. Refer to Figure 3.

When the HIB is opened, it displays a menu of known translators and/or mediators (hereafter referred to as TM's). Each entry of the menu specifies the name of a TM, the site where it can be found, the communication protocol it uses, and other information that may be needed for locating the TM and connecting to it. The user may select any of the TM's on the menu, or the user may enter a new TM not listed.

After a connection is established, an information exchange *session* starts. The user can either type a query directly into the *Active Query* window, or he may select one of the *Frequently-Asked-Queries* shown in the *Queries* window. If a Frequently-Asked-Query is selected, it is copied to the Active Query window. (Typically, these are *fill-in-the-form* queries, so the user must complete the missing parts.) Frequently-Asked-Queries may come from two places: (1) the user may cache previously formulated queries; or (2) the source may provide a list of common queries (we have

[7] In [18] it is argued that user interfaces and browsers will play an important role in exploring heterogeneous information sources.

Figure 3: Querying and Object Browsing

not implemented this feature yet). For example, a translator for Folio may provide templates for finding documents by author, title, and subject, by far the most common queries. The ability to suggest common queries is especially important for "low end" TM's that do not implement the full OEM-QL. In such a case, the user needs guidance as to what queries the source will be able to process.

If a submitted query is valid and successfully executed by the TM, the answer object is returned to the HIB. The user can then navigate through the object structure of the answer. This is better understood if we think of the answer as a tree (or a graph, in the most general case), where the atom objects are the leaves, and the set objects are the internal nodes. Initially, the root and its immediate subobjects are displayed in the object viewer, as illustrated in the left window of Figure 3. Here, the root (label answer) is a set of six documents (label doc). The user can move from the current node to another node by clicking on any of the highlighted direction buttons at the bottom of the window. If a button is not activated, there is no object in that direction. For example, in the left window of Figure 3 one cannot move UP because there are no objects "above" the root. However, the user can move DOWN to the first child of the answer object; the result is shown in the right window of Figure 3. During navigation, the object viewer always shows two levels of the structure (which can be generalized to k levels). Thus, when the current object is a document (label doc) one can see its components, i.e., the TITLE, AUTHOR, and so on (right window). If an atom value is too large to be seen in the viewer (e.g., the abstract of a document), the user can click on it to open a full window that displays the value.

At any time, the user can click on the HELP button to display the "man page" for the label of the current object. As discussed in Section 2, each TM answers the question *What does label* X *mean?* by returning a manual entry. This entry describes in English the meaning of the label and how the value of the object should be interpreted. For example, the entry for the author label

under Folio would explain that names consist of a last name followed by a first name or initials, it would specify the maximum length allowed, it would explain how multiple authors are displayed, and so on. We feel this is a very useful feature of our approach: any time one sees a data value, it is accompanied by a label, and one can immediately find the meaning of the label. This is not only useful to the end-user, but also to the mediator implementor who needs to understand the data that is being integrated or processed.[8]

Notice that the self-describing nature of OEM makes it easy for a user to navigate through unknown objects. If a user knows nothing about a particular source, he can simply pose the query:

```
SELECT ?
FROM root
```

and then browse. As he examines the retrieved labels and their "man pages," he can learn the meaning of each component. Then he can pose more refined queries.

4.2 Translators and Mediators

In this section we illustrate how OEM is used for translation and mediation in the context of our heterogeneous bibliographic information source scenario. The general architecture is shown in Figure 4. Translators are built for all participating bibliographic sources. On top of the translators we use mediators [22] to support objects and queries that are more refined than the objects and queries supported by lower-level translators or mediators. In particular, the mediators directly above the translators reconcile discrepancies between sources (e.g., differences in the structure of objects, the naming of labels, the format of values, etc.), simplifying the task of the mediator that combines information from multiple sources.

To illustrate the operation of the translators and mediators, consider the *Folio* information source and its translator. The Folio translator T receives OEM-QL queries and issues Folio queries. The set of queries $q(T)$ that T is able to translate and execute should have two properties:

1. The translation of any $q(T)$ query into a corresponding Folio query should be as simple as possible, to minimize the translation implementation effort.

2. The set $q(T)$ should preserve as much as possible the power of the underlying query language. Ideally, there should be no Folio query that does not have a corresponding query in $q(T)$.

We have satisfied both properties in the case of Folio by supporting predicates in OEM-QL that correspond directly to the access methods that Folio provides. As an example, Figure 4 shows a typical query entering Folio, asking for the bibliographic entries where the last name of one of the authors is "Ullman" and the first name starts with "J." The corresponding query in OEM-QL is:

[8] The requirement of providing a "man page" for each label could be viewed as a burden, but if the meaning of information is not documented, there is no hope for heterogeneous information access!

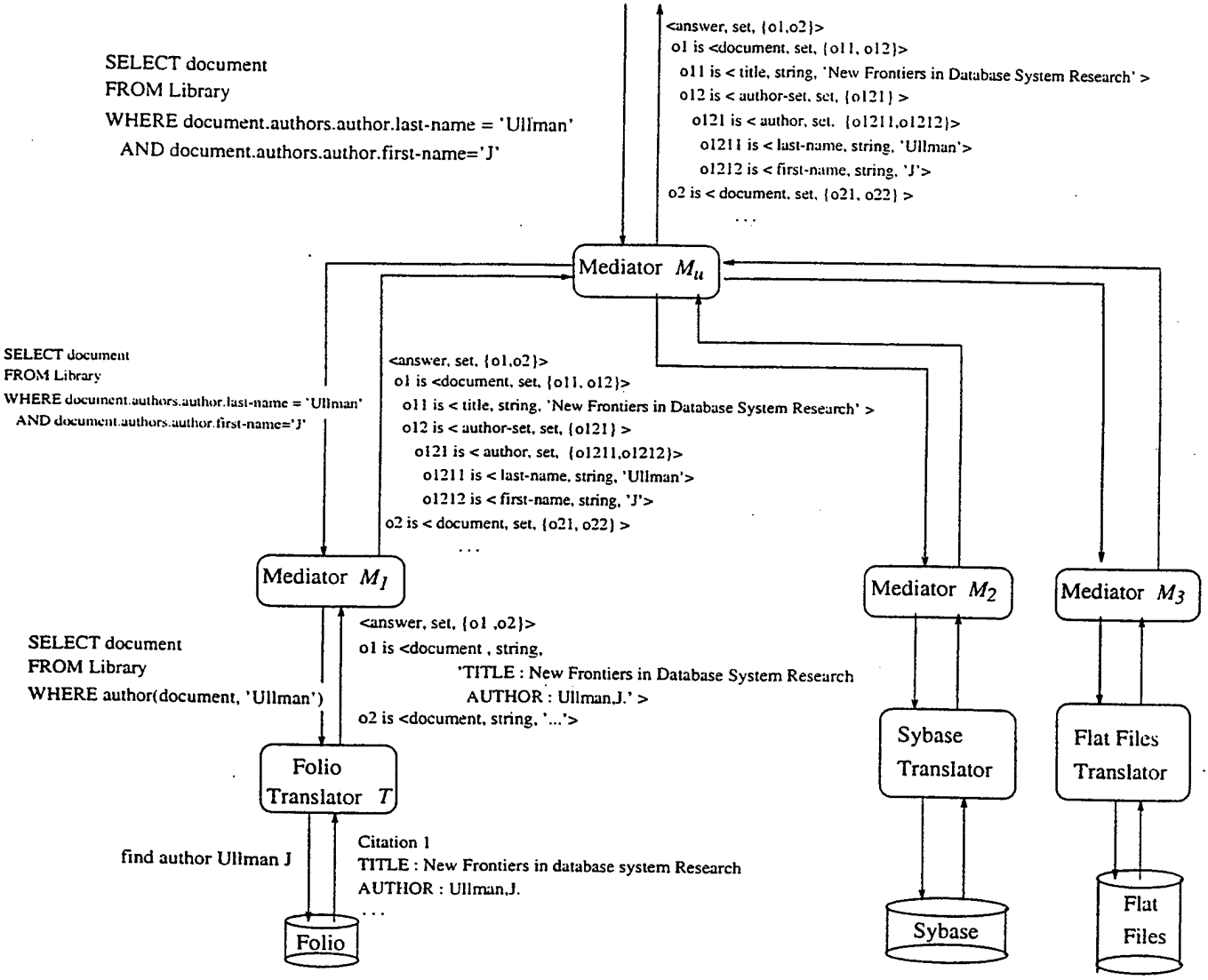


Figure 4: Translation and Mediation Architecture

```
SELECT collection.document
FROM Folio
WHERE author(collection.document, "Ullman J")
```

From this query, T only needs to translate the author predicate to the corresponding author search construct.

As illustrated in Figure 4, translator T uses a straightforward mapping to translate the citations returned from Folio (as a string) into an OEM object. Mediator M_1 refines the structure of the objects exported by T, by extracting the basic components of each bibliographic object (e.g., authors, title). In addition, M_1 supports a wider and more generic set of queries than T. For example, M_1 is able to translate the incoming query shown in Figure 4 to the outgoing one.

A key design criterion here is modularity. Since the translators are likely to be the most complex components (they must deal with the idiosyncrasies of the information sources), our goal is to keep

the work of the translators to a minimum. Once a translator produces its object in some OEM format, additional work can be done by mediators. Note that [7] suggests an average of 6 months effort to implement a translator for a conventional DBMS. In our experience, the total effort can be reduced substantially by shifting work from translators to mediators, and by using the Support Libraries described in Section 5.

The top level mediator M_u in Figure 4 combines the information from several sources into a single document collection. The simplest implementation of this mediator performs a union of all the collections. When M_u receives a query, it effectively "broadcasts" the query to all mediators at lower levels, then merges the answers. Certainly more sophisticated mediation techniques could be useful, such as recognizing and eliminating duplicate results. In the following subsection we describe some initial ideas we have for specifying and implementing mediators.

4.2.1 Mediator Generation

Implementing mediators is a non-trivial task, so our eventual goal is to develop tools for mediator generation. (Similar tools can be used for translator generation, but we focus on mediators here.) The approach described in this section has not yet been implemented, but the ideas are presented to illustrate the type of generators we expect OEM will lead to.

The object translation work of mediator M_1 in Figure 4 (i.e. the "upward" direction) could be described by the following "rule":

```
document replaced by derive_structure(document)
```

This rule specifies that whenever M_1 receives an object O from T, M_1 replaces each (sub)object O_i of O that has label document by a (sub)object O_i' created by derive_structure(O_i). The function derive_structure() may be implemented in a conventional programming language. However, we are currently developing *object-pattern-matching* and *string-pattern-matching* tools that describe object transformations in a high level "label-driven" language. In this way we will often eliminate the need for conventional programming of mediators.

Mediator generators can also describe the process of query translation (the "downward" direction). One approach that easily tackles simple cases relies on templates that describe how predicates (or groups of predicates) in incoming queries are replaced by predicates (or groups of predicates) in outgoing queries. For example, M_1 might use the following query rewriting templates, where X and Y represent variables to be matched:

```
T1. document.authors.author.last-name = "X"
    AND document.authors.author.first-name = "Y"  =>  author(document, "XY")
T2. document.authors.author.last-name = "X"  =>  author(document, "X")
T3. document.authors.author.first-name = "Y"  =>  author(document, "Y")
```

Then, all queries processed by M_1 will be matched against the above templates. For example, if the query:

```
SELECT document
FROM Library
WHERE document.authors.author.last-name = "Ullman"
```

is received by M_1, template T1 will be matched. Variable X will be instantiated to Ullman and the following query will be generated for T:

```
SELECT document
FROM Library
WHERE author(document, "Ullman")
```

As is commonly done in rule systems, our templates may be given an evaluation priority. Assume that T1, T2, and T3 are in decreasing priority. The query received by M_1 in Figure 4 matches all three templates. Since template T1 has highest priority, it is used for the translation shown in the Figure.

5 The OEM Support Libraries

OEM and OEM-QL are designed for a *client* to send queries and obtain corresponding answer objects from a *server*. The server may be a translator or a mediator, while the client may be a mediator or an end-user program (such as the HIB described in Section 4.1). We have implemented general-purpose OEM Support Libraries that provide the common functionality needed for object and query exchange. There are two main components: the *Client Support Library* (CSL) and the *Server Support Library* (SSL).

Figure 5 illustrates how the Support Libraries are used. The implementor of client applications links CSL with the client program in order to create programs with *embedded* CSL calls; CSL calls are used to establish connections with TM servers, to send OEM-QL queries, and to receive OEM objects.[9] CSL procedures handle all low level communications, and deposit retrieved objects in a main memory object buffer. At the server side, the SSL handles incoming connections, buffer management, and management of "slave" processes to execute queries. Note that if a server S obtains its information from another translator or mediator, then S also acts as a client, so it also uses the CSL.

We expect that our Support Libraries will expedite the implementation of mediators, translators, and end-user programs. In addition, implementing these libraries has brought to the surface a number of interesting issues regarding the exchange of objects when one or more participants are not inherently object-oriented. As far as we know, these issues do not arise in conventional, homogeneous object-oriented systems (or at least not in quite this way). Here we discuss one of the most important issues that has arisen, namely that of *partial object fetches.*

[9]Interactive (as opposed to embedded) OEM-QL queries can be posed using the browser described in Section 4.1, which is built on top of the Support Libraries.

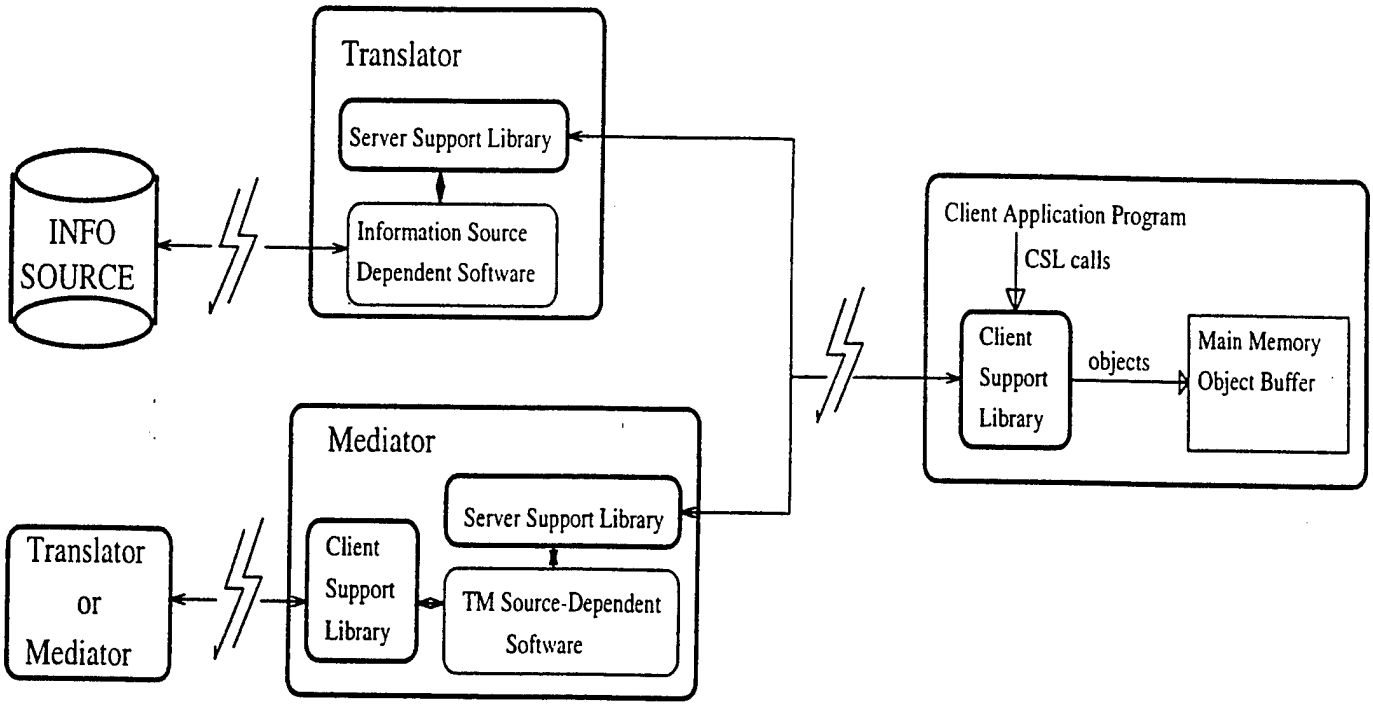


Figure 5: Use of the OEM Support Libraries

In many cases it is extremely inefficient to send the complete answer object to the client in one step. In particular:

1. The client has to wait until the full answer is retrieved from the information source before examining the object. This prevents "pipelined" operation, where the client starts processing subobjects as they arrive. The problem is exacerbated if we have a string of mediators between the source and the client: the client cannot begin processing the answer until all of the intermediate TM's have completed their work.

2. The answer object may be very large. Once a client inspects part of the answer object, the client may determine that it does not need some portions of the answer object, or perhaps does not need the object at all.

To avoid these problems, the Support Libraries provide a *partial fetch* mechanism that enables clients to retrieve only parts of the answer object. The mechanism is used as follows. When the client wishes to request an object, it calls a `query()` function, passing the OEM-QL query as a parameter. The client can then fetch either the full answer object (including subobjects) by calling the `getFullObject()` function, or the client can fetch only the root of the answer object by calling the `getRootObject()` function. In the latter case, additional `getFullObject()` and/or `getRootObject()` calls are used to fetch the subobjects.

Calls to the `getRootObject()` function lead to *incomplete* objects in the client's memory. To illustrate, consider an answer object A whose value is a set of three subobjects, B, C, and D. As discussed in Section 2.1, the copy of A placed in the client's memory can identify its subobjects in

a variety of ways. For example, if subobject B has been fetched to memory, then A will contain a reference to B's memory location. If subobject C is a very large object and the server decides not to transfer it (as in, e.g., the bitmap object described in Section 2.1), then A will contain an OID for C. With partial object fetch there is a third possibility: a subobject, say D, may be "unfetched," i.e. it may be in the server's buffers, or not yet returned by the underlying source. The reference to an unfetched subobject is something that only the Support Libraries understand, and it is specific for the particular call in progress.

Consider what happens when a client wants to examine an unfetched object. One option is to support on-demand retrieval of any unfetched objects. However, this allows the client to traverse answer objects in arbitrary order, implying that the server must cache the entire answer object. Such on-demand fetching would be very difficult for translators such as the one for Folio (recall Section 4). The Folio bibliographic source returns a *stream* of documents, and the translator has no control over the order of the records. For on-demand service, all records would have to be stored by the translator. If the user poses a query that is too broad, the answer object might be enormous.

Consequently, instead of on-demand service, the Support Libraries provides a stream model for retrieving unfetched objects. A "preorder traversal" of the answer object is used, and the client must perform partial fetches in this order. To illustrate, suppose that after a first `getRootObject()` call, the client retrieves an object A whose set value contains three unfetched references, u_1, u_2, and u_3. If the client decides that the number of documents is too large, the client may choose to submit a different query. Otherwise, if the first document is desired, the client issues a `getRootObject()` call with u_1 as a parameter. The first subobject is fetched; suppose it is another set with unfetched references u_{11} and u_{12}. Next the client fetches u_{11}, which happens to be the title of the document. Based on this, the client may decide it wants to skip the rest of the u_1 object. It can do so by issuing a `getRootObject()` call with u_2; this causes the u_1 subobjects that were not fetched to be discarded. Thus, even though the client is constrained to traverse the answer object in a particular order, uninteresting parts can be skipped. At the server side, the uninteresting parts still have to be fetched, but they can be discarded without being transmitted to the client.

Due to space limitations, our description of the OEM Support Libraries and their services has been cursory. Our goal has not been a full description of the Support Libraries, but rather an illustration of the challenging practical issues that arise when there is an "impedance mismatch" between the way an information source provides objects and the way a client wishes to see them. We believe that our Support Libraries provide a general-purpose framework for handling many of these issues.

6 Conclusions and Future Work

We are developing a complete environment and set of tools for integrated access to diverse and dynamic heterogeneous information sources. Exchange of information in our environment is based on the *Object Exchange Model* (OEM) introduced in this paper. OEM retains the simplicity of relational models while allowing the flexibility of object-oriented models. Objects in OEM have

a very simple structure, yet the model is powerful enough to encode complex information. For flexibility, OEM objects are *self-describing*. This approach eliminates the need for regular structure or a predefined schema. However, when structure or schema are present, they can be exploited by OEM translators and mediators.

OEM objects are requested using a declarative query language *OEM-QL*, which is based on nested-SQL query languages. We have found OEM-QL to be both expressive and easy to use. In this paper we have defined the basic constructs of OEM-QL. We are extending the query language along the lines discussed in Section 3. In addition, we plan to add language constructs and underlying support for data modification operations and for *monitors* (or *active rules*).

We have experimented with OEM and OEM-QL by implementing OEM-based access to several quite different bibliographic information sources. Our implementation so far has served a number of purposes:

- It has helped us refine and ratify our design of the model and query language.
- We have uncovered a number of important issues and generic functionalities in the implementation of OEM-based object exchange. This led to our development of the OEM Support Libraries described in Section 5.
- We have realized a need for browsing tools, leading to the Heterogeneous Information Browser described in Section 4.1.
- We have used a layered architecture for translators and mediators (recall Figure 4), which we believe expedites the integration of heterogeneous information sources.

Implementation is currently underway to incorporate additional bibliographic information sources into our system. We are also implementing a translator for the Sybase relational database system, and a browser based on *Mosaic* and the World Wide Web system. Meanwhile, we are beginning to explore techniques for information mediation using OEM. In Section 4.2.1 we described our initial ideas for mediator generation. We plan to refine these concepts to develop a number of useful mediators that combine bibliographic information from multiple sources. We expect that our powerful but simple object exchange model and query language will provide the appropriate platform for quickly achieving this goal.

Acknowledgements

We are grateful to Ed Chang for implementing the Heterogeneous Information Browser, to Ashish Gupta, Laura Haas, and Dallan Quass for valuable comments, and to the entire Stanford Database Group for numerous fruitful discussions.

References

[1] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

[2] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[3] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. In *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, pages 22–29, Kyoto, Japan, April 1991.

[4] R. G. G. Cattell. *Object Data Management.* Addison-Wesley, 1991.

[5] R. G. G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[6] F. Eliasen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20:25–29, 1991.

[7] A. K. Elmagarmid and A. A. Helal. Hetrogeneous database systems. Technical Report TR-86-004, Program of Computer Engineering, Pennsylvania State University, University Park, PA, 1986.

[8] Object Request Broker Task Force. The Common Object Request Broker: Architecture and Specification, December 1993. Revision 1.2, Draft 29.

[9] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

[10] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[11] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[12] H. F. Korth and M. A. Roth. Query languages for nested relational databases. In *Nested Relations and Complex Objects in Databases*, pages 190–204. Springer-Verlag, 1989.

[13] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[14] L. Mark and N. Roussopoulos. Information interchange between self-describing databases. *IEEE Data Engineering Bulletin*, 10(3):46–52, September 1987.

[15] D. S. Marshak. Lotus Notes release 3. *Workgroup Computing Report*, 16:3–28, 1993.

[16] B. Oki et al. The information bus—an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[17] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13:389–417, 1988.

[18] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *Communications of the ACM*, 34:110–120, 1991.

[19] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[20] G. Wiederhold. *File Organization for Database Design.* McGraw Hill, New York, 1987.

[21] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

[22] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, Washington, DC, May 1993.

A Appendix

Here we provide a rigorous specification of the query language that was described informally in Section 3. The syntax of the language is given in the grammar of Figure 6. We discuss two points regarding label variables and predicates, then we define the semantics of queries in our language.

The parenthesized variable following each label in a path is optional. However, if a label L does not include a variable, then L is assigned a variable automatically as follows: L's variable is the concatenation of L's position number in its path together with the name of the nearest specified variable to the left of L; if there is no variable to the left of L then L's variable is L's position number. For example, the path "`bibliography.document(d).topic`" becomes "`bibliography(1).document(d).topic(3d)`". This scheme ensures that two labels in different paths have the same variable if and only if they should refer to the same object component. (Recall from Section 3 that two labels L_1 and L_2 refer to the same object component when they appear in paths that are specified identically from the beginning of the path through L_1 and L_2.) "Wild-card" labels (?) and "wild-path" labels (*) are assigned variables in the same manner.

The predicates that may be specified in a *Condition* are not fixed and may vary from information source to information source, as described in Section 3. Our syntax provides a general notation for arbitrary predicates over multiple arguments. We assume that most information sources support commonly used binary predicates (e.g. equality and inequality over integers), and for convenience we allow these predicates to be written using conventional infix notation, as in the examples of Section 3.

We now define the semantics of an arbitrary query Q. Let O be the object specified in Q's `FROM` clause. We define the semantics in two steps. First we define the set of components of object O that satisfy query Q. Then we define the object A that is returned as the answer to Q. In our definitions we often refer to the label-variable pairs that constitute the paths in Q's `SELECT` and `WHERE` clauses; for brevity we refer to these pairs as *LV*'s.

The first definition formalizes the notion of "path traversals" discussed in Section 3.

Definition A.1 (Valid Binding) A *binding* is a mapping from each LV appearing one or more times in query Q to an object component in O. A binding is *valid* if:

1. Each LV is bound to an object whose label matches the LV's label. If the LV's label is ? or *, then any object label matches.

2. If an LV with a label that is not * appears as the first element on a path, then the LV is bound to object O.

3. Let LV lv_2 follow LV lv_1 on a path. Then lv_1 is bound to an object o of type set or list. If lv_2 does not have label *, then lv_2 is bound to a subobject of o. If lv_2 has label *, then lv_2 is bound to a direct or indirect subobject of o.

4. For each predicate in Q's `WHERE` clause, if each path appearing in the predicate is replaced by the value of the object bound by the last LV on that path, then the predicate is satisfied. □

Query	::=	SELECT *Fetch-Exp* FROM *Object* WHERE *Condition*
Fetch-Exp	::=	*Path* | *Path*.OID
Path	::=	*Label* | *Label*. *Path*
Label	::=	string [(variable)] | ? [(variable)] | * [(variable)]
Object	::=	string /* lexical object identifier */
Condition	::=	true
	|	*Path*
	|	predicate($Value_1$, $Value_2$, ..., $Value_n$)
	|	$Condition_1$ and $Condition_2$
Value	::=	*Path* | constant

Figure 6: Query language syntax

With this Definition we can specify the object components of O that satisfy query Q.

Definition A.2 An object component o *satisfies* query Q if and only if there is a valid binding such that o is bound to the last LV in the path specified in Q's SELECT clause. □

Next we specify the structure of the answer object A that is returned as a result of query Q. We consider two cases separately: (1) when the path specified in Q's SELECT clause does not end with "OID"; (2) when the path specified in Q's SELECT clause does end with "OID".

Definition A.3 (Answer Object: Non-OID) The answer object for a query Q whose SELECT path does not end with "OID" is:

$\langle$answer, set, $\{obj_1, \ldots, obj_n\}\rangle$
obj_1 is $\langle \ldots \rangle$
...
obj_n is $\langle \ldots \rangle$

where obj_1, ..., obj_n are exactly those object components of O that satisfy query Q according to Definition A.2. □

Now suppose query Q's SELECT path does end with "OID". In this case the answer object includes only the identifiers for the relevant object components, and not the objects themselves.

Definition A.4 (Answer Object: OID) The answer object for a query Q whose SELECT path ends with "OID" is:

$\langle$answer, set, $\{OID_1, \ldots, OID_n\}\rangle$

where OID_1, ..., OID_n are the object identifiers for exactly those object components of O that have non-Λ OID's and satisfy query Q according to Definition A.2 □

Flexible Constraint Management for Autonomous Distributed Databases*

Sudarshan S. Chawathe, Hector Garcia-Molina and Jennifer Widom
Computer Science Department
Stanford University
Stanford, California 94305–2140
E-mail: {chaw,hector,widom}@cs.Stanford.edu

1 Introduction

When databases inter-operate, integrity constraints arise naturally. For example, consider a flight reservation application that accesses multiple airline databases. Airline A reserves a block of X seats from airline B. If A sells many seats from this block, it tries to increase X. For correctness, the value of X recorded in A's database must be the same as that recorded in B's database; this is a simple distributed copy constraint. However, the databases in the above example are owned by independent airlines and are therefore autonomous. Typically, the database of one airline will not participate in distributed transactions with other airlines, nor will it allow other airlines to lock its data. This renders traditional constraint management techniques unusable in this scenario. Our work addresses constraint management in such autonomous and heterogeneous environments.

In an autonomous environment that does not support locking and transactional primitives, it is not possible to make "strong" guarantees of constraint satisfaction, such as a guarantee that a constraint is always true or that transactions always read consistent data. We therefore investigate and formalize weaker notions of constraint maintenance. Using our framework it will be possible, for example, to guarantee that a constraint is satisfied provided there have been no "recent" updates to pertinent data, or that a constraint holds from 8am to 5pm everyday. Such weaker notions of constraint satisfaction requires modeling time, and consequently, time is explicit in our framework.

Most previous work in database constraint management has focused on centralized (for e.g., [?]) or tightly-coupled and homogeneous distributed databases (for e.g., [1], [2], [3], [4]). The multi-database transaction approach to constraint management weakens the traditional notion of correctness of schedules [5], [6]. This approach cannot, however, handle a situation in which different databases support different interfaces. In modeling time, our work has similarities with some work in temporal databases [7] and temporal logic programming [8]. Our approach is closer to the event-based specification language in RAPIDE [9].

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

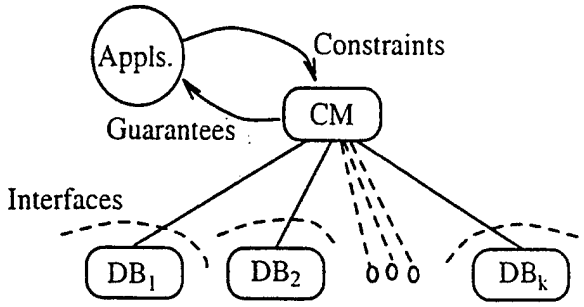


Figure 1: Constraint Management Architecture

In this paper, we give a brief overview of our formal framework for constraint management in autonomous systems and describe the constraint management toolkit we are building. The details of the underlying execution model, semantics of events, and syntax and semantics of the rule language may be found in [10].

2 Formal Framework

In this section, we present an outline of our formal framework for constraint management. Our framework assumes the simplified system architecture shown in Figure 1.[1] Each database chooses the *interface* it offers to the constraint manager (CM) for each of its data items (involved in an inter-database constraint). The interface specifies how each data item may be read, written and monitored by the CM. Applications inform the CM of constraints that need to be monitored or enforced. Based on the constraint and the interfaces available for the data items involved in the constraint, the CM decides on the constraint management *strategy* it executes. This strategy tries to monitor or enforce the constraint as well as possible using the interfaces offered by the local databases. The degree to which each constraint is monitored or enforced is formally specified by the *guarantee*. We describe interfaces, strategies and guarantees below.

2.1 Interfaces

The interface for a data item involved in a constraint describes how that data item may be read, written, and/or monitored by the constraint manager. Interfaces are specified using a notation based on *events* and *rules*. For example, consider a simple write interface for a data item X. This interface promises to write the requested value to X within, say, 5 seconds. We express this as $WR(X,b)@t \rightarrow W_g(X,b)@[t,t+5]$. Here $WR(X,b)@t$ represents a "write-request" event which requests the operation $X \leftarrow b$, and which occurs at some time t. The rule says that whenever such an event occurs, a "write"[2] event, $W_g(X,b)$ occurs at some time in the interval $[t,t+5]$. The interfaces for the data items involved in inter-database constraints are specified by the database administrator of each database, based on the level of access to the database he or she is willing to offer the CM. Currently, we rely on the users of our framework to verify that the interfaces specified do faithfully represent the actual systems.

[1] Note that we assume a centralized constraint manager for simplicity in presentation only; the constraint manager is actually distributed.

[2] $W_g()$ is a *generated* write event which occurs as the result of CM activity, to be distinguished from spontaneous write events, $W_s()$, which occur due to user/application activity in the underlying database.

2.2 Strategies

The strategy for a constraint describes the algorithm used by the constraint manager to monitor or maintain the constraint. Like interfaces, strategies are specified using a notation based on events and rules. In addition to performing operations on data items involved in a constraint, strategies may evaluate predicates over the values of data items (obtained through database read operations) and over private data maintained by the Constraint Manager.

As a simple example, consider the following strategy description, which makes a write request to Y whenever it receives a "notify" event from X:[3] $N(X,b)@t \rightarrow WR(Y,b)@[t,t+7]$.

Once our framework has been used to specify a strategy, and to verify the correctness of a guarantee, then the rule-based strategy specification is implemented using the host language of the Constraint Manager. This is a simple translation, and may be done using a rule engine.

2.3 Guarantees

A guarantee for a constraint specifies the level of global consistency that can be ensured by the Constraint Manager when a certain strategy for that constraint is implemented. Typically a guarantee is *conditional*, e.g., a guarantee might state that if no updates have recently been performed then the constraint holds, or that if the value of a CM data item is true then the constraint holds. Guarantees are specified using predicates over values of data items and occurrences of certain events. For example, consider the following guarantee for a constraint $X = Y$: $(\text{Flag} = \text{true})@t \Rightarrow (X = Y)@@[t-\alpha, t-\beta]$. This guarantee states that if the (Boolean) data item Flag is true at some time t, then $X = Y$ (at all times) during the interval $[t-\alpha, t-\beta]$. Note that this guarantee is weaker than a guarantee that $X = Y$ always, which is very difficult to make in the heterogeneous, autonomous environments we study.

3 A Constraint Management Toolkit

We are building a toolkit that will permit constraint management across heterogeneous and autonomous information systems. For example, this toolkit will allow us to maintain a copy constraint spanning data stored in a Sybase relational database and a file system, or an inequality constraint between a *whois*-like database and an object-oriented database. We give a brief overview of this toolkit in this section.

3.1 Architecture

Figure 2 depicts the architecture of our constraint management toolkit. The Raw Information Source (RIS) is what is already present at each site (for example, a relational database, a file system, or a news feed.) The RISI is the interface offered by each RIS to its users and applications. For example, for a Sybase database, the RISI is based on a particular dialect of SQL, and includes details on how to connect to the server.

The CM-Translator is a module that implements the CM-Interface (the interface discussed in Section 2.1) using the RISI. The CM-RID is a configuration file used to specify (1) which types of CM-Interface (selected from a menu of pre-compiled interface types) are supported by the CM-Translator, and (2) how these interfaces are implemented using the underlying RISI.

[3] A notify event is an event type representing the database containing X notifying the CM of a write to X. Thus $N(X,5)$ means that a write $X \leftarrow 5$ occurred.

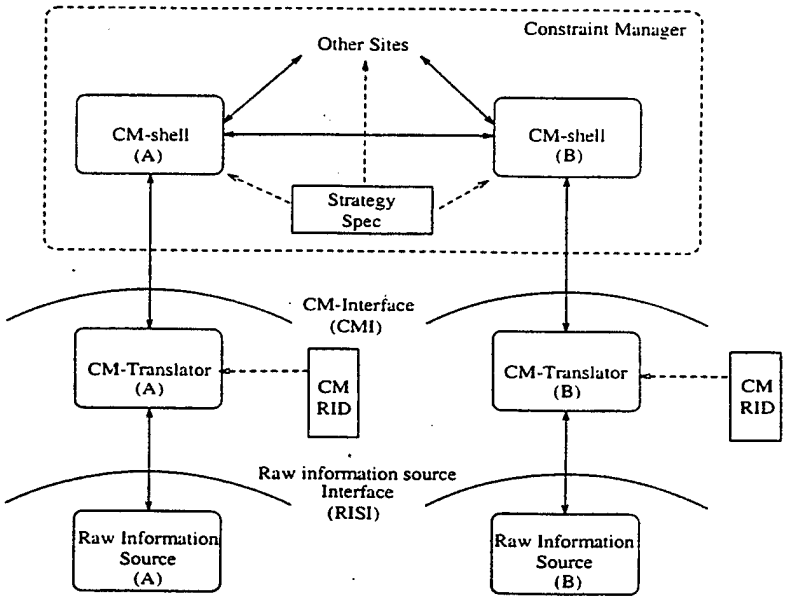


Figure 2: Constraint Management Toolkit Architecture

The CM-Shell is the module that executes the constraint management strategies described in Section 2.2. Since we specify strategies using a rule-based language, the CM-Shells are distributed rule engines that are configured by a Strategy Specification.

3.2 Application

We now describe how database administrators would use our toolkit to set up constraint management across autonomous systems. The database administrators at each site first decide on the CM-Interfaces they are willing to offer, selected from menu of pre-compiled interfaces provided by the toolkit. For example, if the underlying RIS provides triggers, then a notify interface may be offered (where the CM is notified of updates); if not, perhaps a read/write interface can be offered. The choice also depends on the actions the administrator wants to allow. For instance, even if the RIS allows database updates, the administrator may disallow a write interface that lets the CM make changes to the local data.

Each CM-RID file records the interfaces supported, as well as the specification of the RIS objects to which the interface applies. The CM-RID is also the place where site-specific translation information is stored. This includes, for example, the name of the Sybase data server that holds the data and its port number, how a read request from the CM is translated into an SQL query, how a request to set up a notify interface is translated to commands that set up a trigger, and so on.

Next, the administrator uses a Strategy Design Tool (not shown in Figure 2) to develop the CM strategy. This tool takes as input the inter-database constraints, and based on the available interfaces, suggests strategies from its available repertoire. For each suggested strategy, the design tool can give the guarantee that would be offered. The result of this process is the Strategy Specification file, that is then used by the CM-Shells at run time. (Knowledgeable administrators can write their Strategy Specifications, bypassing the Design Tool.)

At run-time, the CM-Shells execute the specified strategy based on the event-rule formalism. The CM-Translators take care of translating events to site-specific operations, and vice versa.

4 Conclusion

We address the problem of constraint management across heterogeneous and autonomous systems. We have argued that this is a problem of practical importance, and that it is not readily amenable to traditional constraint management techniques. Constraint management in autonomous environments requires weaker notions of consistency than those found in literature. We have proposed an event-based formal framework which allows us to express these weaker notions of consistency, and in which time is explicitly modeled. We have also described a constraint management toolkit that we are currently building, which demonstrates some the practical aspects of our work. In the future, we plan to work on expanding our framework to handle more complex interface and constraint types, including those with quantification over data items and probabilities associated with them.

References

[1] Eric Simon and Patrick Valduriez. Integrity control in distributed database systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 621–632, 1986.

[2] Paul Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In *Proceedings of the International Conference on Very Large Data Bases*, pages 581–591, Dublin, Ireland, August 1993.

[3] Stefano Ceri and Jennifer Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the International Conference on Very Large Data Bases*, pages 108–119, Dublin, Ireland, August 1993.

[4] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, 24(12):46–51, December 1991.

[5] Ahmed Elmagarmid, editor. *Special Issue on Unconventional Transaction Management,* Data Engineering Bulletin 14(1), March 1991.

[6] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181, October 1992.

[7] Richard Snodgrass. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[8] Martin Abadi and Zohar Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277-295, 1989.

[9] David C. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 1994.

[10] Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom. Constraint management in loosely coupled distributed databases. Technical report, Computer Science Department, Stanford University, 1993. Available through anonymous ftp from `db.stanford.edu:pub/chawathe/1993/cm-loosely-coupled-dbs.ps`.

A Query Translation Scheme for Rapid Implementation of Wrappers*

Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, Jeffrey Ullman

Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{*yannis,agupta,hector,ullman*}*@cs.stanford.edu*

Abstract

Wrappers provide access to heterogeneous information sources by converting application queries into source specific queries or commands. In this paper we present a *wrapper implementation toolkit* that facilitates rapid development of wrappers. We focus on the query translation component of the toolkit, called the *converter*. The converter takes as input a *Query Description and Translation Language (QDTL)* description of the queries that can be processed by the underlying source. Based on this description the converter decides if an application query is (a) directly supported, i.e., it can be translated to a query of the underlying system following instructions in the QDTL description; (b) logically supported, i.e., logically equivalent to a directly supported query; (c) indirectly supported, i.e., it can be computed by applying a *filter*, automatically generated by the converter, to the result of a directly supported query.

1 Introduction

A *wrapper* or *translator* [C+94, PGMW95] is a software component that converts data and queries from one model to another. Typically, wrappers are used to provide access to heterogeneous information sources, as illustrated in Figure 1.a. In this case, an application (which could be a mediator [Wie92]), issues queries in a single, common query language like SQL. The wrapper for each source converts the query into one or more commands or queries understandable by the underlying source. The wrapper receives the results from the source, and converts them into a format understood by the application.

As part of the TSIMMIS project [PGMW95, GM+] we have developed hard-coded wrappers for a variety of sources, including legacy systems. We have observed, like everyone who has built a wrapper, that writing them involves a lot of effort [A+91, C+94, EH86, FK93, Gup89, LMR90, MY89, T+90]. However, we have also observed that only a relatively small part of the code deals with the specific access details of the source. A lot of code, on the other hand, is either common among wrappers (deals with buffering, communications to the application, and so on) or implements query and data transformations that could be expressed in a high level, declarative fashion.

Based on these observations we have developed a *wrapper implementation toolkit* for rapidly building wrappers. The toolkit contains a library of commonly used functions, such as for receiving

*This work was supported by ARPA Contract F33615-93-1-1339, by NSF IRI 92-23405, by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government.

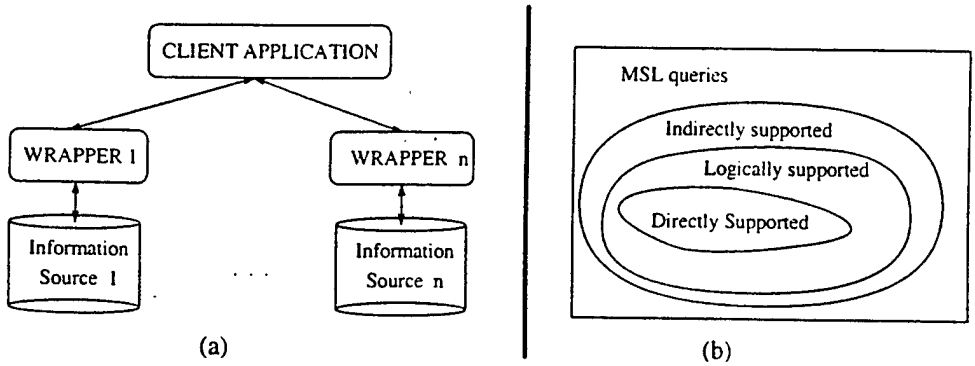


Figure 1: (a) Accessing information through wrappers (b) Supported queries.

queries from the application and packaging results. It also contains a facility for translating queries into source-specific commands and queries, and for translating results into a model useful to the application.

In this paper we focus on the query translation component of the toolkit, which we refer to as the *converter*. (In Section 6 we will describe the other toolkit components and how the converter is integrated with them.) The implementor gives the converter a set of templates that describe the queries accepted by the wrapper. If an application query matches a template, an implementor-provided action associated with the template is executed to produce the *native query* for the underlying source. Note, a native query is not necessarily a string of a well-structured query language (e.g. SQL). In general, the term "native query" may refer to any program used to access and retrieve information from the underlying source.

EXAMPLE 1.1 To illustrate, consider an application that issues SQL queries. One of the sources it accesses has limited functionality, as is true for many sources encountered in a heterogeneous environment. For this illustrative example, assume that the source can only do selection on attribute `dept` of some table, followed by a projection. This ability may be specified as the following template.

```
select $X.$Y from $X where $X.dept=$Z
```

The symbols `$X`, `$Y`, and `$Z` represent *placeholders* that have to be bound to specific constants to produce a valid SQL query. Assume that the following query arrives at the wrapper and is given to the converter:

```
select emp.name from emp where emp.dept='toy'
```

This query matches the template with the bindings `$X = "emp"`, `$Y = "name"`, and `$Z = "'toy'"`. Given the match, the actions associated with the template would then generate the necessary native query to do the actual search on the source. For example, if the underlying source was a file system the actions could produce a "grep" command to search for the string `$Z` in say columns 10-20 of file `$X`. Out of the matching lines, it would return the characters between the string `$Y` and some termination character. □

Example 1.1 illustrates a very simple template matching facility that could be easily implemented using Yacc-like tools [LMB92]. However, since the matching facility is based entirely on string matching, it does not exploit the semantics of the common query language. The following examples show that if converters "understand" queries they are translating, then they can successfully handle many more queries.

EXAMPLE 1.2 Consider the following query template:

```
select $X.$Y from $X where $X.sal=$Z1 and $X.dept=$Z2
```

Syntactically, only queries where the `$X.sal` and `$X.dept` appear in exactly the specified order match this template. The query

```
select emp.name from emp where emp.dept='toy' and emp.sal=100
```

would not match the template. If we wanted to process this type of query we would have to define a second template. In general, we would have to consider an exponential number of orderings of the terms in the `where` clause. It is not practical to have all these templates, especially since all of them would have almost identical actions associated with them. □

EXAMPLE 1.3 Consider a data source that can only do selections on attribute dept and does not understand the notion of projecting out attributes. Such a source can be described with the following template:

```
select * from $X where $X.dept=$Z
```

The following query does not match this template because it includes a projection:

```
select emp.name from emp where emp.dept='toy'
```

However, the wrapper could process the above query by transforming it into one without a projection and then doing the projection on the returned answers. This approach would allow the wrapper to leverage its own capability to handle a much wider class of queries than those specified by the template.

As we will see, our wrapper toolkit can handle this type of query transformation. When the converter is given a query, it generates not only commands for the underlying source, but also a *filter* describing additional processing on the results, if any is required. In our example, the filter would specify a projection over the name attribute. □

In example 1.2 the converter must understand the notion of selection and conjunctive logical expressions. In example 1.3 the converter must understand projections and the fact that a projection over `emp.name` can be obtained a-posteriori from a projection over *. While this knowledge gives the converter the ability to handle more queries, it does mean that the converter must be targeted to a particular incoming query language. Being language specific does not pose a problem for converters because our goal is to develop many wrappers for a given common query language, so it is to our advantage to exploit the features of the common query language. Furthermore, most declarative query languages are based on common principles, so our converter should be easy to modify to other query languages.

Our converters are targeted for the MSL query language [PGMU]. (SQL was only used in our initial examples to motivate our ideas.) MSL is a logic-based language for a simple object-oriented data model, called OEM [PGMW95]. We believe that both OEM and MSL are well suited for integration of heterogeneous information sources. The converter is configured with templates written in the *Query Description and Translation Language* (QDTL). Each template is associated with an action that generates the commands for the underlying source.

Once configured, the converter takes as input an MSL query, and generates commands for the source and a *filter* to be applied to the results. (Actually, in our current design, the converter accepts only a subset of MSL; see Section 2.) The converter will process:

- *Directly supported queries.* These are queries that syntactically match a template.

- *Logically supported queries.* These are queries that produce the same results as a directly supported query. We use the notion of *logical equivalence* to detect queries that fall in this class.

- *Indirectly supported queries.* These are queries that can be executed in two steps: first a directly supported query is executed, and then a filter is applied to the results of the first step. We have appropriately extended the notion of *subsumption* in order to detect the queries that fall in this class.

Figure 1.b graphically shows the types of accepted queries. Although QDTL descriptions syntactically look like Yacc grammars – suitably modified for the description of queries, rather than arbitrary strings – our converter handles a much larger class of queries than the class of directly supported queries that is handled by Yacc. For example, our converter understands the commutativity of a logical conjunction, while Yacc would expect the terms to appear in a specific order. Furthermore, our converter introduces the following innovations:

- A designer can succinctly and clearly define the functionality of each source through a few QDTL templates. Note, a QDTL description is more than a list of "parameterized queries" since it allows the description and translation of infinite sets of queries (See Section 5.)

- The converter, in cooperation with the filter processor, automatically extends the query capabilities of sources that have limited functionality. Note that unlike relational and object oriented databases, where typically all possible queries over the schema are allowed, arbitrary information sources, e.g., legacy systems, permit only limited sets of queries. The automatic extension of query abilities allows us to bring to the same level of functionality different sources and then more easily integrate them.

- The converter, together with the other functions of the toolkit, make it possible to rapidly implement wrappers.

One important thing to notice is that the capabilities of wrappers can be "gracefully extended." That is, one can quickly design a simple wrapper with a few templates that cover some of the desired functionality, probably the one that is most urgently needed. Then templates can be added as more functionality is required.

We start our paper with a brief description of the OEM model and the MSL query language. Then in Section 3 we give a detailed example that shows how QDTL is used. Indirectly supported queries and the notion of query subsumption are further discussed in Section 4, while Section 5 introduces additional powerful QDTL features such as nonterminal templates and metapredicates. In Section 6 we discuss the architecture of wrappers and the wrapper toolkit; we also discuss how the converter is used by the wrapper toolkit to rapidly implement wrappers. Section 7 focuses on the query translation algorithm at the heart of the converter. This is the algorithm that maps input queries to templates and generates filters. The section gives an example-driven description of the algorithm, and the full details can be found in the Appendix A. Finally, Section 8 discusses related work, and Section 9 presents some conclusions and future work. A proof of the correctness of the algorithm can be found at [P+].

2 The OEM Model and the MSL Language

When integrating heterogeneous information sources one often faces unstructured information whose form may change dynamically. Many applications that have to deal with such information use some type of *self-describing* data model where each data item has an associated descriptive

label. Applications include tagged file systems [Wie87], Lotus Notes [Mar93], the Teknekron Information Bus [O+93], LOOM frames [MY89], electronic mail, RFC1532 bibliographic records, and many more. For this reason we have selected a self-describing model, in particular the Object Exchange Model (OEM) [PGMW95], as the common data model exported by our wrappers. OEM captures the essential features of models used in practice, generalizing them to allow arbitrary nesting and to include object identity. OEM does not directly support classes, methods, and inheritance; however, classes and methods can be emulated [PGMW95].

To illustrate OEM, consider the following objects (one object per line):

```
<ob1, person, {sub1,sub2,sub3,sub4,sub5}>
      <sub1, last_name, 'Smith'>
      <sub2, first_name, 'John'>
      <sub3, role, 'faculty'>
      <sub4, department, 'CS'>
      <sub5, telephone, 415-5141292>
```

Each OEM object consists of an *object-id* (e.g., sub4), a *label* that explains its meaning (e.g., `department`), and a *value* (e.g., `'CS'`). Object-id's can be of different types, but for this paper we may think of them as terms that are used to link objects to their subobjects. Labels are strings that are meaningful to the application or the end user. A value can be a scalar such as an integer or a string, or it can be a set of (sub)objects (e.g., the value of the "`person`" object).

At each source, some OEM objects are defined to be *top-level* or root objects. (Of course, the source itself probably does not store OEM objects; this is only the "illusion" created by the wrapper above that source.) Top-level objects provide "entry points" into the object structure from which subobjects can be requested, as explained below.

An application can request OEM objects from a wrapper using the MSL query language [PGMU]. In this paper we will use only a subset of MSL. In particular, we will consider only conjunctive queries that extract a single object together with all its descendants – i.e., direct or indirect subobjects. (In Section 9 we discuss why we make these restrictions.)

To illustrate, consider the following query that searches for top-level `person` objects (i.e., objects with `person` label) containing a `last_name` subobject with value `'Smith'`. The matching objects, together with their `last_name`, `first_name`, ... subobjects, are then retrieved.

```
(Q1) *P :- <P person {<L last_name 'Smith'>}>
```

The query consists of a single *head* and a single *tail* separated by the `:-` symbol. *Variables* are represented by identifiers starting with a capital letter, such as `P` and `L`. The tail describes the search pattern, while the head is the object-id of the objects that will be retrieved.[1] Intuitively, we match the tail pattern against the object structure exported by the wrapper, thereby binding the variables to object components of the wrapper's object structure. The result consists of all the objects (and their descendants) whose object-ids get bound to the variable that appears in the head.

Now we give more details about the matching process. Tails are based on patterns of the form <*object-id label value*>, where each field may be a constant or a variable. When a field (object-id, label, or value) contains a constant then the pattern binds successfully only with OEM objects that have the same constant in the corresponding field. On the other hand, when the field contains a variable the pattern can successfully bind with any OEM object (modulo the restrictions imposed

[1] The * in the head of the query indicates that subobjects are retrieved too. Without the asterisk, a single object is retrieved.

by the other fields in the pattern) and the variable binds to the contents of the corresponding field. If a variable X appears multiple times in a tail, all occurrences of X must bind to the same contents for the tail to successfully bind to an OEM object.

If a pattern A contains a value that has curly braces and more patterns $B, C, \ldots$ inside, then pattern A binds to OEM objects with a set value. The objects that bind to pattern A have one or more subobjects, some of which bind to the patterns $B, C, \ldots$. For example, query Q1 requires that person objects have a last_name subobject with value 'Smith'. Note that we allow the person objects to have subobjects other than last_name as well.

For notational convenience we remove object-id variables from object patterns when the object-id is not useful, i.e. when it appears exactly once in the query. For instance, in query Q1, variable L is not used in the head nor in other parts of the tail. Therefore we can replace the pattern <L last_name 'Smith'> in Q1 by <last_name 'Smith'> without affecting the query. Thus, notationally a pattern with two fields represents a three field pattern with a unique but unspecified variable in the first field.

3 A Detailed Example of Query Translation

We illustrate the use of our converter and QDTL using the following simple example. Say we wish to build a wrapper for a university "lookup" facility that contains information about employees and students. (This example is motivated by an actual service offered by our department at Stanford). The lookup facility is accessed from the command line of computers and offers limited query capabilities. In particular, it can return only the full records of persons, including all fields such as "last name", "first name", and "telephone." There is no way for the user to retrieve only one field, e.g., the telephone number, for a person. Furthermore, the only queries that are accepted by the lookup facility are:

1. Retrieve person records by specifying the last name, e.g.,
   (L2) `lookup -ln Smith`
2. Retrieve person records by specifying the first and the last name, e.g.,
   (L3) `lookup -ln Smith -fn John`
3. Retrieve all person records by issuing the command
   (L4) `lookup`

The queries accepted by the lookup facility can be easily described in our Query Description and Translation Language (QDTL). As discussed in Section 1, a QDTL description consists of a set of templates with associated actions. Below we state description D1 that consists of three *query templates* QT1.1, QT1.2, and QT1.3. For simplicity, we do not yet state the associated actions.

```
(D1) (QT1.1) Query ::= *0 :- <0 person {<last_name $LN>}>
(QT1.2) Query ::= *0 :- <0 person {<last_name $LN> <first_name $FN>}>
(QT1.3) Query ::= *0 :- <0 person V>
```

Each query template appears following the ::= and is a "parameterized query." The identifiers preceded by $, such as $LN and $FN, are *constant placeholders* representing expected constants in the input query. Upper case identifiers, such as 0, are *variable placeholders* denoting variables that are expected at that point in the input query. Note, the variable appearing in the query does not have to have the same name as the template variable.

Each template describes many more queries than those that match it syntactically. More specifically, each template describes the following classes of queries:

- *Directly supported queries.* A query q is directly supported by a template t if q can be derived by substituting the constant placeholders of t by constants and the variables of t by variables. For example, query Q1 is directly supported by template QT1.1 by substituting `O` with `P` and `$LN` with `'Smith'`.

- *Logically supported queries.* A query q is logically supported by template t if q is logically equivalent to some query q' directly supported by t. Two queries q and q' are equivalent if they produce the same result regardless of the contents of the queried source. For example, the following queries are logically supported by template QT1.2 although they are not directly supported:

```
*O :- <O person {<first_name 'John'> <last_name 'Smith'>}>
*O :- <O person {<last_name 'Smith'>}> AND <O person {<first_name 'John>}>
*O :- <O person {<LO last_name 'Smith'>}>
      AND <O person {<LO L V> <first_name 'John'>}>
```

  All these queries are equivalent to the following query Q5, that is directly supported by the template QT1.2:

```
(Q5) *O :- <O person {<last_name 'Smith'> <first_name 'John'>}>
```

- *Indirectly supported queries.* A query q is indirectly supported by a template t if q can be "broken down" into a directly supported query q' and a filter that is applied on the results of q'. We give a definition of indirect support in Section 4; for now we present an example. Consider the following query:

```
(Q6) *Q :- <Q person {<last_name 'Smith'> <role 'student'>}>
```

  This query is not logically supported by any of the templates of description D1. However, our converter realizes that this query is *subsumed* by the directly supported query

```
(Q7) *Q :- <Q person {<last_name 'Smith'>}>
```

  This means that the answer to Q7 contains all the information that is necessary for answering Q6. Thus, the converter matches Q6 to template QT1.1 as if it were Q7, binding `$LN` to `'Smith'` and `O` to `Q`. In addition, the converter generates the filter:

```
*O :- <O person {<role 'student'>}>
```

  The filter is an MSL query that is applied to the result of query Q7 to produce the result of query Q6.

Note, we often say "the description d supports directly, logically, or indirectly the query q" meaning that a template t of d supports directly, logically, or indirectly the query q.

3.1 Formulation of the Native Query

QDTL templates are accompanied by actions that formulate the native queries for the source. For our converter, the actions are written in C, although we could have selected any other language. Let us extend description D1 with actions that formulate native queries such as L2, L3, and L4.

```
(D2) (QT2.1) Query ::= *O:- <O person {<last_name $LN>}>
(AC2.1)                  { sprintf(lookup_query, 'lookup -ln %s', $LN) ;}
(QT2.2) Query ::= *O :- <O person {<last_name $LN> <first_name $FN>}>
(AC2.2)                  { sprintf(lookup_query, 'lookup -ln %s -fn %s', $LN, $FN) ; }
(QT2.3) Query ::= *O :- <O person V>
(AC2.3)                  { sprintf(lookup_query, 'lookup') ; }
```

To illustrate, consider again the input query Q5:

```
*O :- <O person {<last_name 'Smith'> <first_name 'John'>}>
```

This query matches template QT2.2. by binding placeholder `$LN` to `'Smith'` and `$FN` to `'John'`. Then, the action AC2.2 that consists of the C function

```
sprintf(lookup_query, 'lookup -ln %s -fn %s', $LN, $FN)
```

is executed. In this action, `$LN` and `$FN` behave as C variables that at execution time contain the values `'Smith'` and `'John'` respectively. The effect of this action is to write the string `'lookup -ln Smith -fn John'` in the variable `lookup_query`.

This completes the job of the converter on this query. Then, the implementor-provided part of the wrapper takes over, submits the string `lookup_query` to the source and waits for an answer.

4 Query Subsumption

In Section 3 we said that query Q6 was subsumed by Q7 because the former had an additional condition on the "role" subobject. Thus query Q6 selects a subset of the objects obtained by the subsuming query Q7.

A different type of subsumption, specific to object oriented data, occurs when the subsumed query extracts subobjects obtained by the subsuming query. For example, consider the following query Q8 that retrieves the `first_name` subobjects of `person` objects with last name `'Smith'`

```
(Q8) *F :- <O person {<F first_name X> <last_name 'Smith'>}>
```

Query Q8 is subsumed by the following query Q9, that retrieves the full `person` objects of persons with last name `'Smith'` and an unspecified first name.

```
(Q9) *O :- <O person {<F first_name X> <last_name 'Smith'>}>
```

Notice that Q8 and Q9 have exactly the same conditions. However, Q9 subsumes Q8 because the `person` objects retrieved by the latter *contain* the `first_name` objects required by the former. The following definitions formalize the notions we have illustrated.

Definition 4.1 (Object containment) Object O is contained in another object O' if and only if

- Either O and O' are identical, i.e., they have identical object-id, label, and value; or
- O is a subobject (direct or indirect) of O'.

□

Definition 4.2 (Query subsumption) A query q is subsumed by another query q' if each answer object for q is contained in some answer object of q'.[2] □

Definition 4.3 (Indirect support) A query q is indirectly supported by a query q' if

1. q' subsumes q, and
2. there is a filter query f that when applied on the result of q' produces the result of q.

A filter query is formally defined by Definition A.1 in Appendix A. We will say that a template t indirectly supports a query q if t directly supports a query q' that indirectly supports q. □

Note, query subsumption does not necessarily imply indirect support. For example, consider the following query

(Q10) `*F :- <person {<F first_name X>}>`

that subsumes Q8, since it retrieves all `first-name` objects. However, Q10 does not indirectly support Q8, since given a `first-name` object in the result of Q10, we can not tell whether it is a subobject of a `person` with `last_name 'Smith'`.

4.1 Maximal Supporting Queries

Notice that given a query q there may be more than one queries that support q, and these queries may not be logically equivalent. For example, query Q6 on page 7 is supported by query Q7 and also by the query

(Q11) `*O :- <O person V>`

that retrieves all `person` objects.

Note, query Q11 also subsumes query Q7. Thus, Q7 derives fewer unnecessary answers than Q11. From a performance point of view it is better for the wrapper to send Q7 to the source (after the necessary transformation to a native query) rather than Q11, because the former contains more conditions of the original query Q6. Indeed, for our example, query Q7 is the best query directly supported by description D1 that supports query Q6 because Q7 pushes to the source as many conditions as possible. We will say that Q7 is a *maximal supporting query* for Q6.

Definition 4.4 (Maximal supporting query) A query q_s is a maximal supporting query of query q with respect to description d, if

- q_s is directly supported by d,
- q_s indirectly supports q, and
- there is no directly supported query q'_s that indirectly supports q, is subsumed by q_s, and is not logically equivalent to q_s.

□

[2]Note, more general forms of query subsumption may be defined.

Note, there may be more than one maximal supporting query for a given query. For example, assume that a source allows us to place a condition on exactly one subobject of the person objects. This source is specified by the QDTL description (actions not shown):

```
(D3) (QT3.1) Query ::= *O :- <O person {<$L $V>}>
```

For this source, consider input query Q5. This query has two maximal supporting queries:

```
(Q12) *O :- <O person {<last_name 'Smith'>}>
(Q13) *O :- <O person {<first_name 'John'>}>
```

Our converter actually considers all possible maximal supporting queries by considering different ways in which the input query can match the templates of a description. Choosing the optimal maximal subsuming query (when there is more than one) requires knowledge of the contents, semantics, and statistics of the database; our initial implementation does no optimization and simply selects one of the maximal supporting queries. Then, the converter executes the actions associated with that particular maximal query. We give additional details in Section 6.

5 Nonterminals and Other QDTL Features

QDTL allows the use of *nonterminals* to construct grammars that describe more complex sets of supported queries. To illustrate, say that our lookup facility lets us place selection conditions on zero or more of the fields of its records. That is, we can issue commands such as `'lookup -fn John'`, `'lookup -fn John -role faculty'`, `'lookup -role student'`, and so on. Explicitly listing all possible combinations of conditions in our templates would be impractical. (If there are 10 lookup fields, there would be 2^{10} templates.)

With nonterminals, this functionality can be described succinctly. For instance, assuming only three fields, `first_name`, `last_name`, and `role`, we can use the following description (without actions for now):

```
(D4) /* A description with nonterminals */
(QT4.1) Query ::= *OP :- <OP person { _OptLN _OptFN _OptRole}>  /*Query Template*/
(NT4.2) _OptLN ::= <last_name $LN>                          /*Nonterminal template*/
(NT4.3) _OptLn ::=                                  /* empty nonterminal template*/
(NT4.4) _OptFN ::= <first_name $FN>
(NT4.5) _OptFN ::=                                                    /* empty */
(NT4.6) _OptRole ::= <role $R>
(NT4.7) _OptRole ::=                                                  /* empty */
```

Nonterminals are represented by identifiers that start with an underscore (_). Every nonterminal has a *definition* that consists of a set of *nonterminal templates*. For example nonterminal `_OptRole` is defined by nonterminal templates NT4.6 and NT4.7.

A query q is directly supported by a query template t that contains nonterminals if q is directly supported by one of the *expansions* of t. An expansion of t is obtained by replacing each nonterminal n of the query template t with one of the nonterminal templates that define n. For example, the query

```
(Q14) *O :- <O person {<last_name 'Smith'> <role 'professor'>}>
```

is directly supported by template QT4.1 because Q14 matches with the expansion

(E15) *OP :- <OP person {<last_name $LN> <role $R>}>

This expansion is derived from query template QT4.1 by replacing the nonterminal _OptLN with the nonterminal template NT4.2, the nonterminal _OptFN with the nonterminal template NT4.5, and the nonterminal _OptRole with the nonterminal template NT4.6.

5.1 Actions and Attributes Associated with Nonterminals

Nonterminal templates have associated actions, just like query templates. When a query successfully matches with a template, the action for the nonterminal template used during the matching is executed. In addition, every nonterminal n is associated with an *attribute* that is accessible from the templates that use n and the templates that define n. These attributes are similar to the attributes that Yacc (in general context-free grammar parsers) associate with nonterminals, and are used to generate the native query of the underlying source.

Description D4 can be augmented with code to generate the required lookup native query as follows. Note that in the C code, a nonterminal attribute is represented by $ followed by the name of the nonterminal.

```
(D5) (QT5.1) Query ::= *OP :- <OP person { _OptLN _OptFN _OptRole}>
(AC5.1)                 { sprintf(lookup_query, 'lookup %s %s %s', $_OptLN,
                                  $_OptFN, $_OptRole)} ;
(NT5.2) _OptLN ::= <last_name $LN>
(AC5.2)            { sprintf($_OptLN, '-ln %s', $LN) ; }
(NT5.3) _OptLN ::=
(AC5.3)            { $_OptLN = '' ; }
(NT5.4) _OptFN ::= <first_name $FN>
(AC5.4)            { sprintf($_OptFN, '-fn %s', $FN) ; }
(NT5.5) _OptFN ::=
(AC5.5)            { $_OptFN = ''; }
(NT5.6) _OptRole ::= <role $R>
(AC5.6)              { sprintf($_OptRole, '-role %s',$R) ; }
(NT5.7) _OptRole ::=
(AC5.7)            { $_OptRole = '' ; }
```

As discussed earlier, query Q14 is directly supported by description D5. When nonterminal _OptLN matches the <last_name 'Smith'> clause in the query, its associated code is executed, storing the string '-ln Smith' in $_OptLN. Similarly, '-role professor' is stored in $_OptRole. When the query matches template QT5.1, variable lookup_query is assigned the string 'lookup -ln Smith -role professor', which is sent to the lookup facility.

5.2 Recursion

Nonterminal templates may recursively contain nonterminals. This flexibility allows us to describe infinite sets of expansions. The following description – that describes queries with an arbitrary number of conditions on the person subobjects – illustrates recursion

```
(D6) /* This query description involves recursion */
(QT6.1) Query ::= *OP :- <OP person { _Cond }>
(NT6.2) _Cond ::= <$Label $Value> _Cond
(NT6.3) _Cond ::=
```

The query template above directly supports query Q14. To see this we first expand _Cond with the nonterminal template NT6.2, yielding

```
(E7) Query ::= *OP :- <OP person { <$Label $Value> _Cond }>
```

Expanding _Cond again we obtain:

```
(E8) Query ::= *OP :- <OP person { <$Label $Value> <$Label1 $Value1> _Cond }>
```

Note that in the second expansion we replaced the placeholder names with new names `$Label1` and `$Value1`. This policy is essential to avoid confusion with names from other expansions. Finally, we expand _Cond with the nonterminal template NT6.3 (i.e., the "empty" template) to produce an expansion that directly matches query Q14.

In some cases we may want to force placeholder names obtained by expanding nonterminals to be the same as existing placeholder names in the query template. By using parameters as arguments of QDTL nonterminals we can force different templates to refer to the same variable or placeholder (refer to [P+] for details).

5.3 Metapredicates

Descriptions D4 and description D6 accept similar queries, with the exception that D6 accepts any subobject label. For example, D6 will accept the query

```
*P :- <P person {<M fuel 'gasoline'>}>
```

(and an action, not shown in description D6, may translate it into the string `'lookup -fuel gasoline'`) while D4 will not.

We can force D6 to check for particular labels (and effectively schemas) by using *metapredicates*. This capability gives us the same functionality as D4 with a more compact specification. To illustrate, consider the following modification of the description D6:

```
(QT9.1) Query ::= *OP :- <OP person { _Cond }>
(NT9.2) _Cond ::= <$Label $Value> _Cond personsub($Label)
(NT9.3) _Cond ::=
```

The metapredicate `personsub($Label)` checks whether the constant that matches `$Label` is a valid label for some subobject of `person`. The metapredicate `personsub()` is implemented by a C function of the same name. The wrapper implementor provides this function together with description D9.

The converter treats metapredicates simply as additional conditions that must hold for a query to match a template. In our example, after we expand query template QT9.1 with the nonterminal template NT9.2 and then with the nonterminal template NT9.3 we get:

```
*OP :- <OP person {<$Label $Value> personsub($Label)}>
```

Matching this expansion with query Q1 requires that we bind `$Label` to `'last_name'` and `$Value` to `'Smith'`. This binding implies that `personsub('last_name')` must hold. The C function `personsub` is thus invoked, and if it answers "yes" the expansion matches the query.

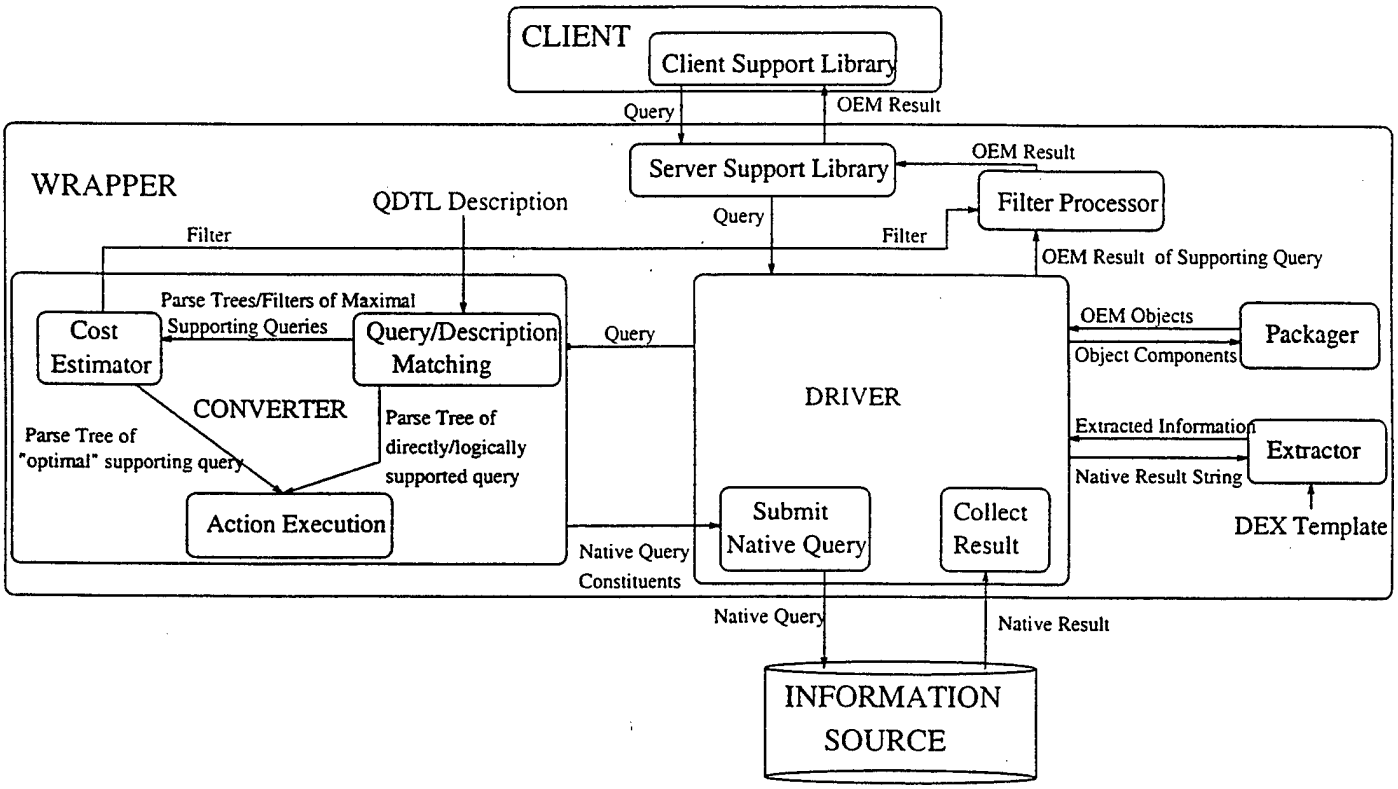


Figure 2: The Architecture of a Wrapper

6 Wrapper Architecture

Figure 2 shows the architecture of the wrappers generated with our toolkit. The shaded boxes represent components provided in the toolkit; the wrapper implementor provides the *driver* that has the primary control of query processing and invokes various services of the toolkit – as is shown in Figure 2. The implementor also provides the QDTL description for the converter, as well as the *Data EXtraction (DEX)* template for the *extractor component* of the toolkit.

Our wrappers behave as servers in a client-server architecture, where the clients are mediators or generic client application programs. Clients use the *client support library* to issue queries and receive OEM results (see Figure 2). The *server support library* component of the toolkit receives queries from the client and dispatches the driver for query processing. The driver invokes the converter, which finds a query that supports the input query and returns the *native query constituents*. The latter are values assigned to variables of the driver that are used to construct the native query. For example, variable `lookup_string` of description D2 contains the only native query constituent for the "lookup" wrapper.

The driver then submits the native query to the underlying information source and receives the result from the source. The driver uses the *extractor* to extract information from the received result and then uses the *packager* to pack the result components into OEM objects. Finally, if during the query/description matching a filter was produced, the driver passes the OEM result and the filter to the *filter processor*.

Subsection 6.1 discusses the converter architecture in more detail. Then, Subsection 6.2 discusses the extractor, while Subsection 6.3 discusses the filter processor.

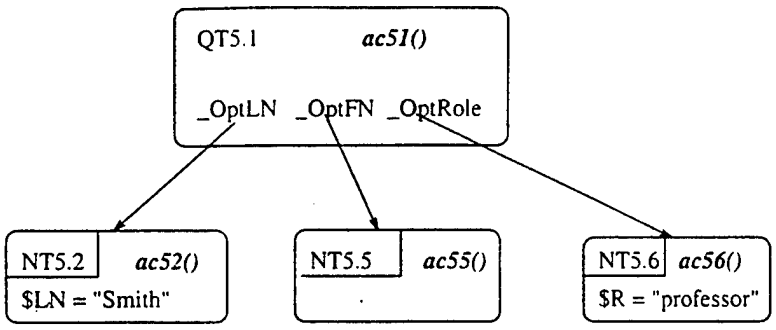


Figure 3: The parse tree

6.1 Converter Architecture

To illustrate, let us assume that the converter is given description D5 that directly supports query Q14 (see Section 5). The *query/description matching* component of the converter produces the parse tree of Figure 3 that contains all the information about the expansions and substitutions obtained while matching the query and the description. The parse tree is used by the *action execution* component of the converter to execute the actions that generate the native query constituents. Note, the converter – unlike the Yacc processor – performs the query/description matching and the action execution in two separate phases because there may be more than one maximal supporting queries, and consequently more than one parse trees. The converter executes actions only after it selects one of the parse trees.

The nodes of the parse tree correspond to the templates that were used for the matching. For readability, in Figure 3 we have named (top left corner) the nodes of the tree using the labels of the corresponding templates in description D5. Also, every node contains a pointer to a C function, such as `ac52()`, `ac55()`, etc, containing the code for the corresponding action. The root node of the parse tree corresponds to query template QT5.1 that matched with the query and points to nodes corresponding to the nonterminal templates – NT5.2, NT5.5, and NT5.6 – that were used. Every node contains a list of the constant placeholders that appear in the template, along with the matching constants.

If there are multiple maximal supporting queries, the query/description matching component passes all the corresponding parse trees to the cost estimator that chooses one of the parse trees either by an *arbitrary choice* or by *cost-based* selection. The later technique assumes that the wrapper has access to cost estimates of the functions provided by the underlying sources, catalog estimates, and so on. In our current implementation, our cost estimator does not perform cost optimization and selects the first parse tree. However, we believe it is important to have the cost optimizer framework in place initially so that optimization may be added later. Once a parse tree is selected, the *action executor* does a postorder traversal of the parse tree and invokes the corresponding action functions. The actions have access to the list of [constant placeholder, matching constant] pairs.

6.2 Information Extraction

Often. legacy systems return data as semi-structured strings. In these cases, the Data Extractor (DEX) can be used to parse the result and identify the required data. DEX is configured with a description of the source's output and information regarding which parts should be extracted. We use a brief example to illustrate how DEX works. Suppose that our sample "lookup" facility returns results as a sequence of text lines, of the form:

```
Record 1
  Last Name: Smith
  First Name: John
  Role: Student
Record 2
  Last Name: ...
```

The goal of the extractor is to extract the `last-name`, `first-name`, and `role` fields of the "lookup" result. This is achieved by giving the following DEX template to the extractor.

```
MATCH STRING (lookup_result)
{ records_number = 0 ;}
( Record # \n
  Last Name\:\ $$(lookup_array[records_number].last_name) \n
  First Name\:\ $$(lookup_array[records_number].first_name) \n
  Role\:\ $$(lookup_array[records_number].role) \n
  { records_number++ ; }
)*
```

Note, inside the `$$(...)` structures appear the names of C variables of the driver. For our running example we may assume that the following data structure has been declared in the driver:

```
struct lookup_type { char[40] last_name ;
                     char[20] first_name ;
                     char[30] role ; } lookuparray[200] ;
```

The above pattern specifies the expected syntax of the string `lookup_result` (that contains the result of lookup), specifies which parts of the output string will be extracted, and in which variables of the driver they will be placed. Our extractor can be viewed as a modification of the Yacc and Lex tools for the more specific problem of information extraction.

6.3 Result Creation and Filter Processing

After the extractor gathers the information in the appropriate data structures of the driver, and the packager constructs the OEM result objects, the *filter processor* applies the filter on the OEM result objects. The filter is produced by the converter while matching the input MSL query with the QDTL description. The filter is an MSL query and is applied to the output of the packager in a 2-step process by the filter processor: First the filter processor creates an algebraic description of the MSL query and then it executes the algebraic description. The algebraic operations can "find the subobjects of an object," "compare the object-id/label/value of an object to a constant," and so on.

7 The Query Translation Algorithm

Answering whether a MSL query q is supported by a QDTL description d is a hard problem. Often we need to reason with descriptions that support infinitely many queries (for instance, description D6). Fortunately, the problem can be reduced to a well-studied problem in deductive database systems. In this section, we discuss how to reduce the "support" problem for QDTL descriptions and MSL queries to a relational context, and we extend existing results from deductive database theory to solve the support problem.

7.1 Correspondence of OEM to Relational Models

In this subsection we discuss how to relationally represent OEM objects, MSL queries, and QDTL descriptions. Note that, the principles of our algorithm can be applied to other object-oriented models as well. For applying our algorithm, the MSL queries and QDTL descriptions are actually converted to relational terms. The objects in the underlying sources are not converted. We discuss how they might be represented relationally to better explain the algorithm.

OEM objects are represented relationally by flattening them into tuples. Each object is represented using tuples of three relations, namely `top`, `object`, and `member`. OEM objects can be converted mechanically to the relational representation using a few straightforward rules: For an object o with object-id `oid`, label `l`, and an atomic value `v`, we introduce the tuple

$$\texttt{object(oid, l, v)}$$

If o is a set object with object-id `oid` and label `l`, then we introduce the tuple

$$\texttt{object(oid, l, set)}$$

Assuming that o has subobjects $o_i, 1 \leq i \leq n$, identified by $\texttt{oid}_i$, $1 \leq i \leq n$ we introduce n tuples

$$\texttt{member(oid, oid}_i\texttt{)}$$

where $1 \leq i \leq n$. Finally, if o is a top-level object identified `oid`, we also introduce the tuple

$$\texttt{top(oid)}$$

The relational representation of MSL queries is obtained similarly by querying the `top`, `object`, and `member` relations that represent the object structure referenced in the query.

EXAMPLE 7.1 Consider the query

```
*O :- <O person {<LM last_name 'Smith'>}>
```

The above query selects top-level objects `O`, i.e., the subgoal `top(O)` must hold. Object `O` is a `person` set-object, i.e., the subgoal `object(O, person, set)` must hold. `O` must have a subobject identified by `LM`, i.e. `member(O, LM)` must hold. Finally, `LM` must be a `last_name` object with atomic value `'Smith'`, i.e., `object(LM, last_name, 'Smith')` must hold. We collect all the object-id's `O` that satisfy the stated conditions into a relation `answer`. Thus, the MSL query can be written as the following datalog query:

```
answer(O) :- top(O), object(O, person, set),
             member(O,LM), object(LM, last_name, 'Smith')
```

□

The general algorithm for converting an MSL query to a relational form is given in [P+]. A similar algorithm for translating a QDTL description to a relational description is described in [P+]. We illustrate the translation via an example.

EXAMPLE 7.2 Consider description D6 from Subsection 5.2. The equivalent relational representation is:

```
(R10) Query ::= answer(OP) :- top(OP), object(OP, person, set), _Cond(OP)
      _Cond(OP) ::= member(OP, OS), object(OS, $Label, $Value), _Cond(OP)
      _Cond(OP) ::=
```

Note, the nonterminal `_Cond` has been replaced by the nonterminal `_Cond(OP)` that has one parameter. We need this parameter because we have to denote that object `OS` that appears in the nonterminal template associated with `_Cond` is a subobject of `OP`. We associate with every template of the relational representation, the action of the corresponding template of the original QDTL template. □

7.2 Algorithm

In this section we illustrate the algorithm that for a given MSL query written relationally, finds maximal supporting queries from a QDTL description also written relationally. If the query is indirectly supported, the algorithm derives the filter MSL query that needs to be applied to the OEM objects picked by the underlying source.

First we illustrate the process of finding a supporting query given the description D and the query Q. Then we show how description D can be expressed as a (possibly recursive) Datalog program $P(D)$. We show that the problem of determining if a description D supports query Q, is the same as the problem of determining if program $P(D)$ contains[3] (subsumes) query Q and a corresponding filter query exists. Thus, a supporting query is found in two steps: (a) find a subsuming query, and (b) find the corresponding filter. We extend an existing algorithm that checks containment (from Section 14.5 of [Ull89]), to answer step (a). We refer to the containment algorithm from [Ull89] as QinP. We extend the algorithm to handle step (b).

Algorithm QinP gives a yes/no answer to the containment question and thus to the subsumption question. Thus, we further extend the algorithm to find the actual maximal supporting queries, the corresponding filters, and also the native query constituents for the underlying source. We describe in detail the extended algorithm *X-QinP* in the Appendix. We continue with examples to illustrate the required extensions.

EXAMPLE 7.3 (Finding Supporting Queries) This example illustrates, in relational terms, how to find supporting queries for a MSL query from a QDTL description. We use this example in the rest of this subsection.

Consider the query Q16 that selects all person objects that have a subobject with label last_name and value 'Smith':

```
(Q16)  answer(O) :- top(O), object(O, person, set), member(O,N),
                    object(N, last_name, 'Smith')
```

Consider the description D11 that supports queries that select person objects that have at least one subobject that has a specified label and a specified value.

```
(D11) (QT11.1)   Query ::= answer(P) :- top(P), object(P, person, set), _Cond(P)
      (NT11.1)   _Cond(P) ::= member(P,X), object(X,$L,$V)
```

By expanding template QT11.1 using nonterminal expansion rule NT11.1 we obtain expansion (E17).

```
(E17):  answer(P) :-top(P), object(P, person, set), member(P,X), object(X,$L,$V)
```

(E17) is identical to query Q16 by substituting appropriately variables and placeholders. Thus, $D11$ directly supports $Q16$.

Alternatively, consider query Q18 that picks person objects with specified values of subobjects last_name and ssn.

```
(Q18)  answer(O) :- top(O), object(O, person, set), member(O,L),
                    object(L, last_name, 'Smith'), member(O,S),
                    object(S, ssn, '123')
```

[3] A query Q is contained in a program P if for all databases, P derives a superset of the answers derived by Q.

Description D11 does not directly support query Q18 because the query imposes selection conditions on two subobjects whereas the description supports queries with only single subobject selections. However, E17 produces two queries that indirectly support Q18:

- E19 enforces the selection condition on subobject `last_name`.

  ($E19$): `answer(O) :- top(O), object(O, person, set), member(O,L), object(L, last_name, 'Smith')`

- E20 enforces the selection condition on subobject `ssn`.

  ($E20$): `answer(O) :- top(O), object(O, person, set), member(O,S), object(S, ssn, '123')`

□

As illustrated above, nonterminals in a query template are expanded to yield expansions of the query template that match the query of interest. If a nonterminal is defined using a recursive template, then the query template has an infinite number of expansions. To find a supporting query requires checking if query Q matches one or more of the infinite number of expansions.

In the next section we show how to reduce the problem of finding a supporting query in a description to the problem of determining whether a conjunctive query is contained in a Datalog program. We extend a known solution to the latter problem to find all the supporting queries, the corresponding filter queries, and the corresponding native query constituents.

7.2.1 Expressing Descriptions as Recursive Datalog Programs

In description D11, if we replace the query template with the rule defining predicate `answer`, and replace `::=` with `:-` in the nonterminal template NT11.1, then we get a Datalog program that uses constant placeholders in addition to variables and constants.[4] The constant placeholders are similar to variables except that they match a subset of the constants which the variables match, and placeholders are used in the actions that produce the native query constituents. We use $P(D)$ to refer to the Datalog program corresponding to description D. The process of finding an expansion of a query template in a description D that matches a target query Q, is the same as determining if the Datalog program $P(D)$ produces a rule E that defines predicate `answer` and matches query Q. Rule E matches query Q if the head of E maps to the head of Q, and each subgoal of E maps to some subgoal of Q (with appropriate restrictions on how to map variables, placeholders, and constants). Query $Q16$ and expansion (E17) in Example 7.3 illustrated this case.

Note, in our framework both Q and E are *conjunctive* queries [Ull89] extended with placeholders. From existing work on the containment of Datalog queries we know that the existence of a mapping from E to Q is a necessary and sufficient condition for the containment of E in Q.[5] Thus, the problem of determining if a description D supports a conjunctive query Q is the same problem as determining if some rule produced by Datalog program $P(D)$ contains query Q (modulo the existence of a filter query). Furthermore, for Datalog this question is the same as asking if the program $P(D)$ contains Q. Algorithm QinP from [Ull89] answers exactly this question.

7.2.2 Applicability and Extensions of Algorithm QinP

First, we illustrate how the containment algorithm QinP finds subsuming queries given a query and a description. Then we illustrate the extensions that need to be made to Algorithm QinP.

[4] Templates with empty expansions are handled as explained in the Appendix.

[5] The containment results used in this paper, hold in the presence of constant placeholders.

EXAMPLE 7.4 (Applying Algorithm QinP) Consider query Q18 from Example 7.3.

```
(Q18)  answer(O) :- top(O), object(O, person, set), member(O,L),
                    object(L, last_name, 'Smith'), member(O,M),
                    object(M, ssn, '123')
```

and the description D11

```
answer(P)  :-  top(P), object(P, person, set), Cond(P)
Cond(P)    :-  member(P,X), object(X,$L,$V)
```

To determine if program P(D11) contains query Q18 Algorithm QinP does the following: First the algorithm "freezes" Q18, i.e., it replaces each variable in each subgoal of Q18 by a corresponding "frozen" constant and puts the resulting frozen facts in a database DB(Q18). The frozen constant for a variable is represented by a constant of the same name in lower case and with a bar on it. The over-bars distinguish frozen constants from regular constants.

```
top(ō), object(ō, person, set), member(ō,l̄), object(l̄, last_name, 'Smith'),
member(ō,m̄), object(m̄, ssn, '123')
```

Then, the program P(D11) is evaluated on DB(Q18) to check if the program derives the frozen head of Q18, namely "answer($\bar{o}$)". If yes, then it is the case that the program contains the query.

While evaluating the program on the frozen database, constant placeholders in P(D11) are assigned only regular constants and not frozen constants, because frozen constants correspond to variables in the target query. Variables in P(D11) are assigned either frozen or regular constants. □

The above example illustrates that Algorithm QinP gives only a yes/no answer to the subsumption question. That is, if program $P(D)$ derives the frozen head of query Q then we know that D subsumes Q. However, the algorithm does not find the particular subsuming query (for instance, $(E19)$ in Example 7.3). The algorithm does not find the selection conditions that are not enforced by each subsuming query (for instance, $(E19)$ does not enforce `ssn = '123'`). Finally, algorithm QinP does not retain enough information to build the native query constituents. Algorithm X-QinP provides this functionality and finds all the maximal supporting queries (if there are multiple such queries). We illustrate these points via a set of examples.

EXAMPLE 7.5 (Multiple Subsuming Queries) Example 7.3 shows that query Q18 is indirectly supported by Description D11 (page 17) via two subsuming queries $(E19)$ and $(E20)$. We discuss in more detail how to obtain $(E19)$.

```
(E19):  answer(O) :- top(O), object(O, person, set), member(O,L),
                     object(L, last_name, 'Smith')
```

$(E19)$ is obtained by algorithm X-QinP, because program P(D11) derives the frozen head of query Q18 using frozen base facts `top(`$\bar{o}$`)`, `object(`$\bar{o}$`,person,set)`, `member(`$\bar{o}$`,`$\bar{l}$`)`, and `object(`$\bar{l}$`,last_name,'Smith')`. $(E20)$ is obtained similarly. As guaranteed by extended algorithm X-QinP, $(E19)$ and $(E20)$ are *maximal.* □

Note, in Example 7.5 the subsuming queries $(E19)$ and $(E20)$ do not use all the frozen facts obtained by freezing the target query Q18. Facts not used to derive a subsuming query correspond to unenforced selection conditions and constitute the *residue* for that query. For instance, for

subsuming query ($E19$) the frozen facts member($\bar{o}$,$\bar{s}$) and object($\bar{s}$,ssn,'123') constitute the residue. A non-empty residue implies that the subsuming query does not enforce all the selection conditions of the input query. Thus, we need to formulate a filter MSL query that when applied to the OEM objects picked by the subsuming query, gives the same result as the input query. A filter query may not always exist as illustrated by the following example.

EXAMPLE 7.6 (Existence of a Filter query) Consider a query Q that for all persons with last_name 'Smith' picks the subobject corresponding to the first_name. Consider a query template T that picks the first_name subobjects of all persons. Algorithm X-QinP infers that T generates a query Q_s that subsumes Q along with the residue member(P,LN), object(LN, last_name, 'Smith'), i.e., the parent objects of the picked first_name subobjects have last_name value 'Smith'. This unapplied selection condition cannot be enforced on the result of query Q_s because there is no way to infer from the result what first_name is associated with which last_name. Thus, no filter query exists for query Q_s. Algorithm X-QinP discards subsuming queries for which no filter query may be formulated. For instance, we discard a subsuming query if its residue refers to an object that is not a subobject of the result of the subsuming query. We also discard queries based on other criteria described in the Appendix.

Algorithm X-QinP generates filter queries for subsuming queries that are retained and thus are supporting queries. A conservative filter query may consist of all the conditions in the input query that can be applied on the result of the supporting query. In this case, some conditions may be redundant. Our algorithm derives optimal filter queries, that is, removes all redundant conditions. Below we illustrate the filter MSL query produced by the algorithm for query ($E19$).

```
*O :- <O person {<S, ssn, '123'>}>
```

□

The last extension to algorithm QinP handles the actions that are executed by the converter to generate the native query constituents. The actions are associated with the nonterminal and query templates of a description D. When we reduce a query template or nonterminal template T of a decription D into a rule R of the datalog program $P(D)$ we associate with R the action that is associated with the template T. Then, the problem of executing the actions associated with the templates of D reduces to the problem of executing the actions associated with the corresponding rules of $P(D)$. Algorithm X-QinP tracks the rules used to derive a supporting query and subsequently executes the actions associated with these rules to produce the native query constituents.

7.3 Performance of X-QinP

In the worst case, X-QinP is exponential in the number of conditions in the query plus the number of templates in the description. Nevertheless, in many practical cases X-QinP is polynomial. For example, if both the query and the templates have explicitly specified labels and there is no recursive template (e.g., description D4) X-QinP needs time proportional to the product of the query size and the number of templates. Furthermore, we expect the number of conditions and templates to be relatively small, so running time should be acceptable.

8 Related Work

Integration of heterogeneous information sources has attracted great interest from the database community[Wie92, LMR90, T+90, Gup89, A+91, C+94, FK93]. Significant work has been done

on integrating and querying data that is in the same model as the integration system. However, underlying sources may have different data models, thus making necessary the existence of wrappers, and consequently, the facilitation of the wrapper construction. [EH86] points out that typically the construction of a wrapper requires "6 month work". Indeed, there are existing techniques for translating schemas and queries of a data model A (say, relational) to schemas and queries of a data model B (say, an object-oriented data model)[QR95, A+91]. Our query translation methodology is different from the above cited work in two ways:

1. We provide a toolkit that can translate queries from our common data model to queries of *any* data model, i.e. we are not bound to a specific "target" data model. Note, the underlying information sources may even not have a well-defined data model.

2. We assume that the source may have limited query capabilities, i.e., not every query over the schema of the underlying source can be answered.

We contribute in two ways to the problem of limited query capabilities (that has been recently recognized [RSU, C+94] as being very important in integration of arbitrary heterogeneous information sources): First, we provide a concise language for description of query capabilities. Second, we automatically increase the query capabilities of a source.

The problem of finding a supporting query is related to the problem of determining how to answer a query using a set of materialized views in place of some of the base relations used by the query [LY85, L+, RSU]. This work uses a fixed set of prespecified views to answer a query. However, we use an infinite set of views that are specified via templates. The templates can specify views like "all relations obtained by applying a single selection predicate to any relation," thus not requiring that the relation name be known. Alternatively, arbitrary numbers of selection conditions can be specified, thereby allowing in the set of "available" views, views that have arbitrarily long specifications. Another difference from [LY85, L+, RSU] is that our focus is on object oriented views and queries and not relational views even though we use some of the same tools, like containment.

9 Conclusions

In this paper we have presented a toolkit that facilitates the implementation of wrappers. The heart of the toolkit is a converter that maps incoming queries into native commands of the underlying source. The converter provides the translation flexibility of systems like Yacc, but giving substantially more power, i.e. translating a much wider class of queries.

The wrapper toolkit is currently under implementation, with the server support library, extractor, and packager already built and tested. These components, with hard-wired converters, have been used to build wrappers for Sybase, a collection of BiBTex files stored on a Unix file system, and a bibliographic legacy system. We are currently implementing the QDTL configurable converter described in this paper; it should be operational by the summer of 1995.

In the future, we plan to extend the power of QDTL descriptions and of the converter to handle a larger class of queries. The currently handled class of conjunctive MSL queries will be extended by using a containment checking algorithm more general than algorithm QinP. Also, we plan to extend the algorithm to detect when multiple queries of the underlying source together support the given application query.

Acknowledgements

We are grateful to Jennifer Widom, Andreas Paepcke, Vasilis Vassalos, and the entire Stanford Database Group for numerous fruitful discussions and comments.

References

[A+91] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

[C+94] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[EH86] A. K. Elmagarmid and A. A. Helal. Heterogeneous Database Systems. TR-86-004, Program of Computer Engineering, Pennsylvania State University, University Park, 1986.

[FK93] J.-C Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. In *Advanced database systems*. N.R Adam, B.K. Bhargava (editors), (ISBN 3-540-57507-3) Springer-Verlag, 1993, pages 313-36.

[GM+] H. Garcia-Molina et al. The TSIMMIS Approach to mediation: Data models and languages (extended abstract). To appear in 1995 NGITS workshop. Also available by ftp at `db.stanford.edu` as `pub/garcia/1995/tsimmis-models-languages.ps`.

[Gup89] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases*. IEEE Press, 1989.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[L+] A.Y. Levy et al. Answering queries using views. To appear in *PODS*, 1995.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[LY85] P.A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB Conf.*, pages 259–69, 1985.

[Mar93] D. S. Marshak. Lotus Notes release 3. *Workgroup Computing Report*, 16:3–28, 1993.

[MY89] R. MacGregor and J. Yen. LOOM:integrating multiple AI programming paradigms. *Proc. Intl. Joint Conf. on Artificial Intelligence*, August 1989.

[O+93] B. Oki et al. The Information Bus—an architecture for extensible distributed systems. In *Proc. of the 14th ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, 1993.

[P+] Y. Papakonstantinou et al. A query translation scheme for rapid implementation of wrappers (extended version). Available by ftp at `db.stanford.edu` as the file `/pub/papakonstantinou/1995/querytran-extended.ps`.

[PGMU] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. Available by ftp at `db.stanford.edu` as the file `/pub/papakonstantinou/1995/medmaker.ps`.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Data Engineering Conf.*, pages 251–60, March 1995.

[QR95] X. Qian and L. Raschid. Query interoperation among object-oriented and relational databases. In *Data Eng. Conf.*, pages 271–9, 1995.

[RSU] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. To appear in PODS 95. Also available by ftp at `db.stanford.edu` as `pub/rajaraman/1994/limited-opsets.ps`.

[T+90] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, New York, 1989.

[Wie87] G. Wiederhold. *File Organization for Database Design*. McGraw Hill, New York, 1987.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

Appendix

A Extended Algorithm QinP

A.1 Extended Algorithm

Now we state a variant of algorithm QinP. Algorithm QinP gives a yes/no answer to the containment question and thus to the support question, modulo the existence of a filter query. We extend the algorithm to find the maximal supporting queries, to construct the corresponding filter queries, and and to construct the corresponding parse trees.

In particular, we extend and modify the algorithm QinP in the following ways:

1. we keep track of which specific expansion of the Datalog program actually contains the query and thus infer the conditions that constitute the residue for the expansions,

2. we keep track of the *implied equalities*. An implied equality arises when we map a variable to a constant. For example, consider the query

   (Q21) `answer(O) :- top(O), object(O,L,V)`

   that supports the query

   (Q22) `answer(O) :- top(O), object(O,person,V)`

   Note, we have to filter the result of Q21 to keep only the objects with label `person`. We will say that the corresponding filter has to check the implied equality `L = person`. Thus, we keep the subgoal `object` of Q22 in the residue, though it maps to the `object` subgoal of Q21.

3. we find "maximal" expansions that have as many conditions of the target query as is possible given the description,

4. we relax the condition that the head of the expansion is the same as the query head to allow the head of the expansion to represent a parent object of the query head,

5. we check that the residue conditions can be evaluated, and

6. we construct the filter that evaluates them.

The algorithm X-QinP follows four basic steps (there are comments in the algorithm that indicate the start of each step):

- **Step 1:** Find the queries with minimal residue with respect to the input query.
- **Step 2:** Select the maximal subsuming queries, i.e. the minimal residue queries that pick objects that contain the required objects.
- **Step 3:** Select the maximal supporting queries, i.e. check the existence of an appropriate filter query for every selected maximal subsuming query.
- **Step 4:** For every maximal supporting query construct an optimal filter query, in the sense that the constructed filter query has as few conditions as possible.

Note, in order to simplify the description of our algorithm we do not include metapredicates and we do not describe the execution of actions.

Input

Conjunctive Query Q where head is of the form $answer(X)$

Description $P(D)$ (recursive Datalog program that defines $answer$ and uses EDB $member$, $object$, top)

Output

A set of maximal supporting queries, associated filters, and associated parse trees

Method

Minimize the query Q (see [Ull89]), i.e. remove all rendundant subgoals

Freeze the query Q – replace each variable A with a constant $\bar{a}$

% Start of Step 1 : Computation of minimal residue instances

% add the frozen facts to DB along with the set of underlying facts and implied equalities

For each ground fact f obtained from the frozen body of Q add to DB

the five-tuple $< f, U, I, A, P >$ where

U = $\{f\}$ *% set of underlying facts for f*

I = $\{\}$ *% set of implied equalities used to derive f*

A = $\{\}$ *% set of residue facts resulting from I*

P = f *% parse tree associated with the fact*

The five-tuple $< f, U, I, A, P >$ is called an "instance of fact f".

% Apply the rules of P(D) to the facts in DB to generate all possible ground facts

% along with their underlying facts, implied equalities, and parse trees

For all rules that have an empty body, "$h(\bar{H}) : -$"

Add the fact $< h(c), \{\}, \{\}, \{\}, nil >$ to DB for all constants and frozen constants c in DB.

Loop

For $1 \leq i \leq k$ where k is the number of rules in description $P(D)$ do

Let rule r_i be:

$h(\bar{H}) : -p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$

where $\bar{X}$ is the set of variables and placeholders in $\bar{H} \cup \bar{X}_1 \cup \ldots \cup \bar{X}_n$

For each assignment θ that:

1. maps variables V in $\bar{X}$ to constants and frozen constants
2. maps every placeholder V in $\bar{X}$ to a constant
3. there exists a vector $[t_1, \ldots, t_n]$ such that $t_j =< \theta(p_j(\bar{X}_j)), U_j, I_j, A_j, P_j >$ and t_j is in DB

do *% derive "optimal" instances of $\theta(h(\bar{H}))$*

Initialize sets I_{temp} and A_{temp} to $\{\}$.

For each variable V in $\bar{X}$ that θ maps to a constant

Add the mapping $V \rightarrow \theta(V)$ to I_{temp} *% Add an implied equality*

Find a $\theta(p_j(\bar{X}_j))$, such that $V \in X_j$, insert $\theta(p_j(\bar{X}_j))$ in A_{temp}

% For each valid instantiation of rule r_i, add an instance of a fact to DB

(A) For every vector $[t_1, \ldots, t_n]$ where $t_j =< \theta(p_j(\bar{X}_j)), U_j, I_j, A_j, P_j >$ and t_j is in DB

Let $t_{new} =< f_{new}, U_{new}, I_{new}, A_{new}, P_{new} >$ =

$< \theta(h(\bar{H})), \bigcup_{j=1}^{n}(U_j), \bigcup_{j=1}^{n}(I_j) \cup I_{temp}, \bigcup_{j=1}^{n}(A_j) \cup A_{temp}, node(r_i, [P_1, \ldots, P_n]) >$

For all $t \in DB$ of the form $< f_{new}, U_t, I_t, A_t, P_t >$

% Discard t_{new} if it uses fewer subgoals and

% *has more implied equalities than some* $t \in DB$
If $U_{new} \subseteq U_t$ and $I_{new} \supseteq I_t$
continue with next iteration of (A)
% *Discard* t *if it uses fewer subgoals and has*
% *more implied equalities than* t_{new}
If $U_t \subset U_{new}$ and $I_t \supset I_{new}$
Remove t from DB
% *Add "better" or incomparable new instances*
Add t_{new} to DB
Until no new instances of facts are derived

% *Step 2: Find all maximal subsuming queries*
For each instance $t =< f, U_t, I_t, A_t, P_t >$ in DB such that
assuming that $\texttt{answer}(\bar{x})$ is the frozen head of query Q, either
$f = \texttt{answer}(\bar{x})$, or
$f = \texttt{answer}(\bar{y})$ and there is a sequence of member facts
$\texttt{member}(\bar{x}, \bar{s}_1), \ldots, \texttt{member}(\bar{s}_n, \bar{y})$, i.e. $\bar{y}$ is reachable from $\bar{x}$
$residue(t)$ = ((subgoals in frozen $tail(Q)$) minus U_f) union A_f

% *Step 3: Check if an appropriate filter* f *exists for the query* q *represented by* t
% *if* f *exists then* q *is a maximal supporting query*
if $t =< \texttt{answer}(\bar{w}), U_t, I_t, A_t, P_t >$ satisfies the following conditions
1. for every subgoal $\texttt{object}(\bar{z}, L, V)$ or $\texttt{member}(\bar{z}, \bar{z}')$ there is a sequence of member facts
$\texttt{member}(\bar{w}, \bar{s}_1), \ldots, \texttt{member}(\bar{s}_n, \bar{z})$, $n \geq 0$, i.e. $\bar{z}$ is reachable from $\bar{x}$
2. there is no frozen constant $\bar{v}$ that appears in more than two subgoals such that
an instance of $\bar{v}$ appears in $residue(t)$ and
another instance of $\bar{v}$ is not reachable from $\bar{w}$ via member facts

% *Step 4: Construct filter and maximal subsuming query*
For each instance $t =< f, U_f, I_f, A_f, P_f >$ do
Initialize store to be the empty set
For each subset S of $body(Q)$ such that S is a superset of $residue(t)$ do
if Q is equivalent to "$head(Q) : -NV(U_f), S$" then
% NV *replaces each frozen constant* $\bar{x}$ *with a unique variable* X' *except*
% *the argument of* f *that is replaced by the unfrozen variable* X
add S to store
Eliminate all $S \in$ store if $\exists S' \in$ store such that $S' \subseteq S$
For each remaining $S \in$ store
output query $head(Q) : -f, S$ as the filter query.
else discard t

Definition A.1 (Filter of a query q_s with respect to input query q) Assume that q_s defines the predicate $\texttt{answer}_s$ and q defines the predicate answer. A filter q_f of q_s wrt q is any query of the form

$$\texttt{answer}_f(X) : -\texttt{answer}_s(Y), \langle cond(Y) \rangle$$

where $\langle cond(Y) \rangle$ is a set of subgoals member and object such that

- every subgoal $\texttt{member}(S_p, S_c)$ of $\langle cond(Y) \rangle$ is reachable from Y.

- every subgoal $\texttt{object}(O, label, value)$ of $\langle cond(Y)\rangle$ is reachable from Y, and
- $\texttt{answer}_f(x)$ holds if and only if $\texttt{answer}(x)$ also holds

□

Definition A.2 (Indirect support of a query q by a query q_s) A client query q is indirectly supported by a query q_s if there is a filter of q_s with respect to q. □

Definition A.3 (Minimal residue instance) Any instance $t =< f_t, U_t, I_t, A_t, P_t >$ is called minimal residue instance if there is no $t' =< f_{t'}, U_{t'}, I_{t'}, A_{t'}, P_{t'} >$ such that $U_t \subset U_{t'}$ and $I_{t'} \subset I_t$. □

Extracting Semistructured Information from the Web

J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo

Department of Computer Science
Stanford University
Stanford, CA 94305-9040

{hector,joachim,cho,aranha,crespo}@cs.stanford.edu
http://www-db.stanford.edu/

Abstract

We describe a configurable tool for extracting semistructured data from a set of HTML pages and for converting the extracted information into database objects. The input to the extractor is a declarative specification that states where the data of interest is located on the HTML pages, and how the data should be "packaged" into objects. We have implemented the Web extractor using the Python programming language stressing efficiency and ease-of-use. We also describe various ways of improving the functionality of our current prototype. The prototype is installed and running in the TSIMMIS testbed as part of a DARPA I^3 (Intelligent Integration of Information) technology demonstration where it is used for extracting weather data form various WWW sites.

1. Introduction

The World Wide Web (WWW) has become a vast information store that is growing at a rapid rate, both in number of sites and in volume of useful information. However, the contents of the WWW cannot be queried and manipulated in a general way. In particular, a large percentage of the information is stored as static HTML pages that can only be viewed through a browser. Some sites do provide search engines, but their query facilities are often limited, and the results again come as HTML pages.

In this paper, we describe a configurable extraction program for converting a set of hyperlinked HTML pages (either static or the results of queries) into database objects. The program takes as input a specification that declaratively states where the data of interest is located on the HTML pages, and how the data should be "packaged" into objects. The descriptor is based on text patterns that identify the beginning and end of relevant data; it does not use "artificial intelligence" to understand the contents. This means that our extractor is efficient and can be used to analyze large volumes of information. However, it also means that if a source changes the format of its exported HTML pages, the specification for the site must be updated. Since the specification is a simple text file, it can be modified directly using any editor. However, in the future we plan to develop a GUI tool that generates the specification based on high-level user input.

The vast majority of information found on the WWW is *semistructured* in nature (e.g., TSIMMIS [1], LORE [2], Garlic [3], Information Manifold [4], RUFUS [5]). This means that WWW data does not have a regular and static structure liké data found in a relational database. For example, if we look at classified advertisements on the Web, the "fields" and their nesting may differ across sites. Even at a single site, some advertisements may be missing information, or may have extra information. Because of the semistructured nature of WWW data, we have implemented our extractor facility so that it outputs data in OEM (Object Exchange Model) [1] which is particularly well suited for representing semistructured data. OEM is the model used by our TSIMMIS (The Stanford IBM Manager of Multiple Information Sources) project. Thus, one of our TSIMMIS wrappers [6] can receive a query targeted to a set of HTML pages. The wrapper uses the extractor to retrieve the relevant data in OEM format, and then executes the query (or

whatever query conditions have not been applied) at the wrapper. The client receives an OEM answer object, unaware that the data was not stored in a database system.

In this paper, we describe our approach to extracting semistructured data from the Web using several examples. Specifically, we illustrate in detail how the extractor can be configured and how a TSIMMIS wrapper is used to support queries against the extracted information.

2. A Detailed Example

For our running example, let us assume that we have an application that needs to process weather data, such as temperature and forecast, for a given city. As one of its information sources, we want to use a Web site called IntelliCast [7] which reports daily weather data for most major European cities (see Figure 1).

		Tue, Jan 28, 1997		*Wed, Jan 29, 1997*	
country	*city*	*forecast*	*hi/lo*	*forecast*	*hi/lo*
Austria	Vienna	snow	-2/-7	snow	-2/-7
Belgium	Brussels	ptcldy	3/-4	ptcldy	3/-4
Czech Republic	Prague	snow	-1/-7	snow	-1/-7
Denmark	Copenhagen	fog	3/-1	fog	3/-1
England	Birmingham	ptcldy	9/-3	ptcldy	7/3
England	Liverpool	ptcldy	8/2	ptcldy	6/2
England	London	ptcldy	9/0	ptcldy	8/4
England	Manchester	ptcldy	8/-1	ptcldy	6/3
England	Plymouth	ptcldy	9/3	ptcldy	8/5

Figure 1: A snapshot of a section of the IntelliCast weather source.

Since this site cannot be queried directly from within another application (e.g., "What is the forecast for Vienna for Jan. 28, 1997?") we first have to extract the contents of the weather table from the underlying HTML page[1] which is displayed in Figure 2.

2.1 The Extraction Process

Our *configurable extraction program* parses this HTML page based on the specification file shown in Figure 3. The specification file consists of a sequence of *commands*, each defining one extraction step. Each command is of the form

[variables, source, pattern]

where *source* specifies the input text to be considered, *pattern* tells us how to find the text of interest within the source, and *variables* are one or more extractor variables that will hold the extracted results. The text in variables can be used as input for subsequent commands. (If a variable contains an extracted URL, we can also specify that the URL be followed, and that the linked page be used as further input.) After the last command is executed, some subset of the variables will hold the data of interest. Later we describe how the contents of these variables are packaged into an OEM object.

[1] The line numbers shown on the left-hand side of this and the next figures are not part of the content but have been added to simplify the following discussions.

```
 1 <HTML>
 2 <HEAD>
 3 <TITLE>INTELLICAST: europe weather</TITLE>
 4 <A NAME="europe"></A>
 5 <TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=509>
 6 <TR>
 7 <TD colspan=11><I>Click on a city for local forecasts</I><BR></TD>
 8 </TR>
 9 <TR>
10 <TD colspan=11><I> temperatures listed in degrees celsius </I><BR></TD>
11 </TR>
12 <TR>
13 <TD colspan=11><HR NOSHADE SIZE=6 WIDTH=509></TD>
14 </TR>
15 </TABLE>
16 <TABLE CELLSPACING=0 CELLPADDING=0 WIDTH=514>
17   <TR ALIGN=left>
18     <TH COLSPAN=2><BR></TH>
19     <TH COLSPAN=2><I>Tue, Jan 28, 1997</I></TH>
20     <TH COLSPAN=2><I>Wed, Jan 29, 1997</I></TH>
21   </TR>
22   <TR ALIGN=left>
23     <TH><I>country</I></TH>
24     <TH><I>city</I></TH>
25     <TH><I>forecast</I></TH>
26     <TH><I>hi/lo</I></TH>
27     <TH><I>forecast</I></TH>
28     <TH><I>hi/lo</I></TH>
29   </TR>
30   <TR ALIGN=left>
31     <TD>Austria</TD>
32     <TD><A HREF=http://www.intellicast.com/weather/vie/>Vienna</A></TD>
33     <TD>snow</TD>
34     <TD>-2/-7</TD>
35     <TD>snow</TD>
36     <TD>-2/-7</TD>
37   </TR>
38   <TR ALIGN=left>
39     <TD>Belgium</TD>
40     <TD><A HREF=http://www.intellicast.com/weather/bru/>Brussels</A></TD>
41     <TD>fog</TD>
42     <TD>2/-2</TD>
43     <TD>sleet</TD>
44     <TD>3/-1</TD>
45   </TR>
       .
       .
   </TABLE>
     .
</HTML>
```

Figure 2: A section of the HTML source file.

Looking at Figure 3, we see that the list of commands is placed within the outermost brackets '[' and ']', and each command is also delimited by brackets. The extraction process in this example is performed by five commands. The initial command (lines 1-4) fetches the contents of the source file whose URL is given in line 2 into the variable called `root`. The '#' character in line 3 means that everything (in this case the contents of the entire file) is to be extracted. After the file has been fetched and its contents are read into `root`, the extractor will filter out unwanted data such as the HTML markup commands and extra text with the remaining four commands.

The second command (lines 5-8) specifies that the result of applying the pattern in line 7 to the source variable `root` is to be stored in a new variable called `temperature`. The pattern can be interpreted as follows: "discard everything until the first occurrence of the token </TR> ('*' means discard) in the second table definition and save the data that is stored between </TR> and </TABLE> ('#' means save)." The two <TABLE tokens between the '*' are used as navigational help to identify the correct </TR> token since there is no way of specifying a numbered occurrence of a token (i.e., "discard everything until the *third* occurrence of </TR>"). After this step, the variable `temperature` contains the information that is stored in lines 22 and higher in the source file in Figure 2 (up to but not including the subsequent </TABLE> token which indicates the end of the temperature table).

```
1  [["root",
2    "get('http://www.intellicast.com/weather/europe/')",
3    "#"
4   ],
5   ["temperatures",
6    "root",
7    "*<TABLE*<TABLE*</TR>#</TABLE>*"
8   ],
9   ["_citytemp",
10   "split(temperatures,'<TR ALIGN=left>')",
11   "#"
12  ],
13  ["city_temp",
14   "_citytemp[1:0]",
15   "#"
16  ],
17  ["country,c_url,city,weath_tody,hgh_tody,low_today,weath_tomorrow,hgh_tomorrow,low_tomorrow",
18   "city_temp",
19   "*<TD>#</TD>*HREF=#>#</A>*<TD>#</TD>*<TD>#/#</TD>*<TD>#</TD>*<TD>#/#*"
20  ]]
```

Figure 3: A sample extractor specification file.

The third command (lines 9-12) instructs the extractor to split the contents of the `temperature` variable into "chunks" of text, using the string `<TR ALIGN=left>` (lines 22, 30, 38, etc. in Figure 2) as the "chunk" delimiter. Note, each "chunk" represents one row in the temperature table. The result of each split is stored in a temporary variable called `_citytemp`. The underscore at the beginning of the name `_citytemp` indicates that this is a temporary variable; its contents will not be included in the resulting OEM object. The split operator can only be applied if the input is made up of equally structured pieces with a clearly defined delimiter separating the individual pieces. If one thinks of extractor variables as lists (up until now each list had only one member) then the result of the split operator can be viewed as a new list with as many members as there are rows in the temperature table. Thus from now on, when we apply a pattern to a variable, we really mean applying the pattern to *each* member of the variable, much like the `apply` operator in Lisp.

In command 4 (lines 13-16), the extractor copies the contents of each cell of the temporary array into the array `city_temp` starting with the second cell from the beginning. The first integer in the instruction `_citytemp[1:0]` indicates the beginning of the copying (since the array index starts at 0, 1 refers to the second cell), the second integer indicates the last cell to be included (counting from the end of the array). As a result, we have excluded the first row of the table which contains the individual column headings. Note, that we could have also filtered out the unwanted row in the second command by specifying an additional `*</TR>` condition before the '#' in line 7 of Figure 3. The final command (lines 17-20) extracts the individual values from each cell in the `city_temp` array and assigns them into the variables listed in line 17 (`country`, `c_url`, `city`, etc.).

After the five commands have been executed, the variables hold the data of interest. This data is packaged into an OEM object, shown in Figure 4, with a structure that follows the extraction process. OEM is a schema-less model that is particularly well-suited for accommodating the semistructured data commonly found on the Web. Data represented in OEM constitutes a graph, with a unique root object at the top and zero or more nested subobjects. Each OEM object (shown as a separate line in Figure 4) contains a label, a type, and a value. The label describes the meaning of the value that is stored in this component. The value stored in an OEM object can be atomic (e.g., type *string, url*), or can be a set of OEM subobjects. For additional information on OEM, please refer to [8].

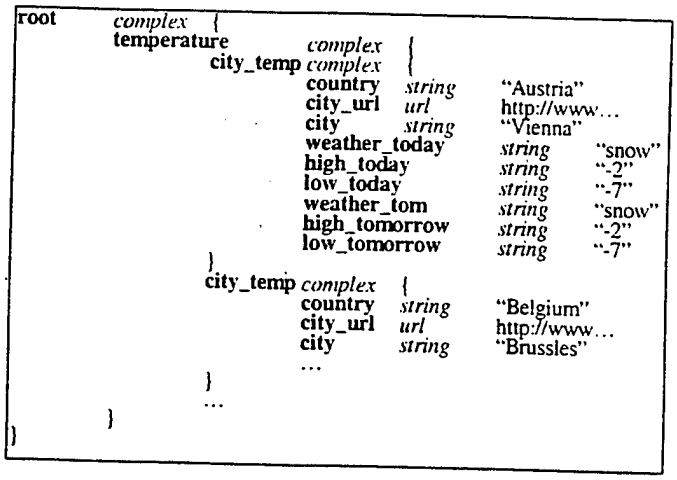


Figure 4: The extracted information in OEM format.

Notice that the sample object in Figure 4 reflects the structure of our extractor specification file. That is, the root object of the OEM answer will have a label `root` because this was the first extracted variable. This object will have a child object with label `temperature` because this was the second variable extracted. In turn, the children are the `city_temp` objects extracted next, and so on. Notice that variable `_citytemp` does not appear in the final result because it is a temporary variable.

2.2 Customizing the Extraction Results

As discussed in the previous section, the outcome of the extraction process is an OEM object that contains the desired data together with information about the structure and contents of the result. The contents and structure of the resulting OEM object are defined in a flexible way by the specification file. For instance, we could have chosen to extract additional data, and to create an OEM result that has a different structure than the one shown in Figure 4. For example, we can also extract the date values in lines 19 and 20 of Figure 2. Then, we can group together the temperature and weather data that is associated with each date, creating an OEM object such as the one depicted in Figure 5. Although not shown in our example, we could have also specified that different label names be used in the OEM object than those that are used for the extraction variables.

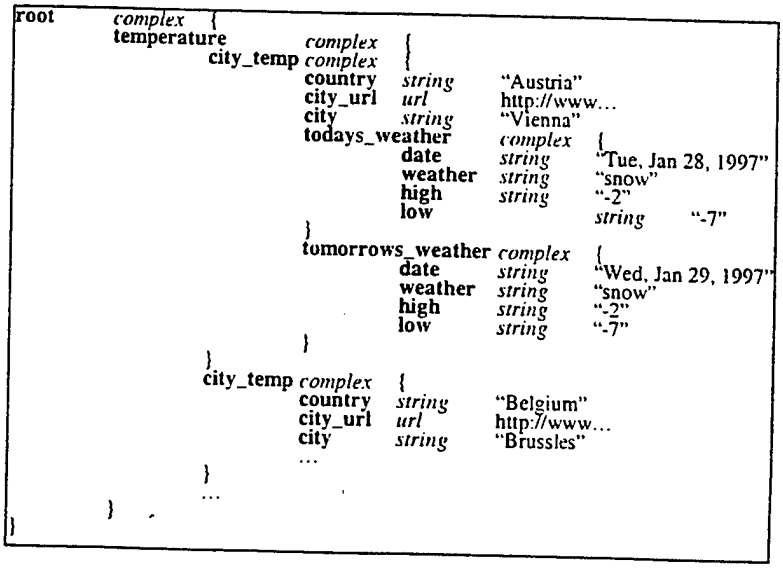


Figure 5: A different OEM result object.

It is important to note that there may be several different ways of defining the individual extraction steps that ultimately result in the same OEM answer object. Thus when defining the specification file one can proceed in a way that is most intuitive rather than worrying about finding the only "correct" set of steps. For instance, in our example, we could have avoided the usage of the temporary array `_citytemp` by filtering out the unwanted header information in the previous step. However, both approaches ultimately lead to the same result[2] (with slight differences in performance).

2.3 Additional Capabilities

In addition to the basic capabilities described in the previous section, our extractor provides several other features and constructs that simplify the extraction steps and at the same time enhance the power of the extractor. For example, the `extract_table` construct allows the automatic extraction of the contents of an HTML table (i.e., the data that is stored in each of its rows) as long as the table can be uniquely identified through some patterns in the text (this would allow us to collapse steps 2 and 3 in our example). An other useful operation is the `case` operator that allows the user to specify one or more possible patterns that are expected to appear in the input. This is especially useful for sources where the structure of the file is dynamic (in addition to the actual data). If the first pattern does not match, the parser will try to match each of the alternate patterns until a match has been found. If none of the patterns match, the parser will ignore the rest of the current input and continue parsing the data from the next input variable (if there is one).

As a last example of the extraction capabilities of our parser, consider the frequent scenario where information is stored across several linked HTML pages. For example, one can imagine that the weather data for each city is stored on its own separate Web page connected via a hyperlink. In this case, one can simply extract the URL for each city and then obtain the contents of the linked page by using the `get` operator as shown in the second line of Figure 3.

2.4 Querying the Extracted Result

In order to allow applications to query the extracted results, we need to provide a queryable interface that can process queries such as "What is the high and low temperature for Jan. 29 for Vienna, Austria?" and return the desired result (e.g., "high: -2, low -7"). Rather than developing a new query processor from scratch, we decided for now to reuse the wrapper generation tools that we have developed in the TSIMMIS project. With this toolkit, we can generate wrappers that have the ability to support a wide variety of queries that are not natively supported by the source, in our case the extracted output (for more details on wrappers see [6, 9]). With this approach, we only need to provide a simple interface that accepts a request for the entire extracted result (this is equivalent to supporting a query such as "`SELECT * FROM ...`"). Making use of the wrapper's internal query engine, we can indirectly support most of the commonly asked queries (i.e., selections and projections on the extracted output). Currently, applications can interact with TSIMMIS wrappers using one of two query languages (LOREL[3] [2] and MSL[4] [10]). Wrappers return results in OEM format.

In the future, we plan to store extracted OEM results in LORE (Lightweight Object Repository) to make use of its more sophisticated query processor that has been optimized for answering (LOREL) queries on semistructured data. In addition, we will get the ability to cache extracted results which will allow us to reuse already extracted information and provides independence from unreliable sources.

---

[2] We chose the approach described here since it demonstrates the additional capabilities of the extractor but the solution without the temporary variable is more efficient.

[3] LOREL (LORE Language)is a query language that was developed at Stanford as part of the LORE (Lightweight Object Repository) project for expressing queries against semistructured data represented in OEM.

[4] MSL (Mediator Specification Language) is a rule-based language, which was developed as part of the TSIMMIS project for querying OEM objects.

3. Evaluation

An important design goal when developing our extractor was to find the right balance between its inherent capabilities on one hand and ease of use on the other. We think we have achieved both, which becomes apparent when one compares our approach to other existing tools such as YACC [11] or PERL [13] regular expressions, for example. Although YACC is a more powerful and more generic parser, a YACC grammar for extracting Web data from the HTML source in Figure 2 would be much more complex and difficult to generate (after all, writing a YACC specification is nothing else than writing a program using a much more complex language). For example, YACC does not support hierarchical splitting, an important feature of our extractor that demonstrates its close relationship to the hierarchical organization of Web data and simplifies the specification that a user has to write.

We have also considered using an existing HTML parser which is natively available in Python[5]. This HTML parser "understands" SGML syntax and can automatically identify HTML tags in the input stream. Upon encountering an HTML tag, the parser executes a user-defined function using the tag attributes as function input. Thus, it is very easy to extract text and HTML tags from an input file. However, the native parser does not understand the semantic connections that exist between some of the tags (i.e., begin and end tags, list members, etc.) and cannot easily be used for building an OEM object that preserves the hierarchical structure of the data. All of the processing and construction has to be handled by externally invoked user-defined functions. In addition, although this native parser is extremely flexible in terms of input processing capabilities, it is not as efficient in terms of raw processing speed as our own parser which is implemented on top of the Python `find` command[6].

A drawback of our approach is that the extraction mechanism depends on outside (human) input for describing the structure of HTML pages. This becomes an issue when the structure of source files changes rapidly requiring frequent updates to the specification file. Using a different approach, Ashish et al. [12], attempt to insert machine learning techniques into their extraction program for automatically making intelligent guesses about the underlying HTML structure of a Web site. Their approach is aimed at eliminating most of the user intervention from the extraction process. By contrast, the approach that we are pursuing here is two-pronged and relies on human intelligence supported by a flexible extractor program: (1) we are enabling our extractor to exit gracefully from a variety of cases in which the underlying structure does not match the specification, and (2), we are making the process of writing the extraction specification itself as easy and efficient as possible. In the future, we intend to develop a GUI for helping users generate and maintain correct specification files. Our planned interface will resemble a Web browser in that it can render the marked-up HTML source. More importantly however, it will enable the user to simply "highlight" the information that is to be extracted directly on the screen without having to write a single line of specification (which will be generated automatically by the GUI).

4. Conclusion

There has been much interest recently in moving data from the WWW into databases, of one type or another. This way, data that is embedded in HTML documents can be searched more effectively, and can be better managed. Our extractor is a flexible and efficient tool that provides a currently missing link between a lot of interesting data (which resides on the Web) and the applications (which have no direct access to the Web data).

We are currently using the extractor in our TSIMMIS testbed for accessing weather and intelligence data from several Web sites. As a result of our initial tests, we implemented the `case` operator as described in Sec. 2.3. The need for such an operator arose since we frequently encounter minor irregularities in the structure of the underlying HTML pages (e.g., weather data that is temporarily missing for a given city,

[5] Our extractor is implemented using Python [14] version 1.3.

[6] Comparison tests have shown a processing speed of 10K/sec. of text for the native HTML parser vs. 2 MB/sec. for the Python `find` command.

etc.) from which our first prototype did not recover. We are now in the process of extracting other kinds of semistructured information from the Web and find that the currently implemented set of extraction operators is powerful enough to handle all of the encountered sources; i.e., our specification files are straightforward and easy to understand.

References

[1] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," In *Proceedings of Tenth Anniversary Meeting of the Information Processing Society of Japan*, Tokyo, Japan, 7-18, 1994.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel Query Language for Semistructured Data," In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[3] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, A. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, "Towards heterogeneous multimedia information systems: the Garlic approach," In *Proceedings of Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, Los Angeles, California, 123-130, 1995.

[4] T. Kirk, A. Levy, J. Sagiv, and D. Srivastava, "The Information Manifold," AT&T Bell Laboratories, Technical Report 1995.

[5] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas, "The RUFUS System: Information Organization for Semi-Structured Data," In *Proceedings of Nineteenth International Conference on Very Large Databases*, Dublin, Ireland, 97-107, 1993.

[6] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman, "A Query Translation Scheme for Rapid Implementation of Wrappers," In *Proceedings of Fourth International Conference on Deductive and Object-Oriented Databases*, Singapore, 1995.

[7] Weather Services International. "INTELLICAST: Europe Weather." URL, http://www.intellicast.com/weather/europe/.

[8] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object Exchange Across Heterogeneous Information Sources," In *Proceedings of Eleventh International Conference on Data Engineering*, Taipei, Taiwan, 251-260, 1995.

[9] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, and R. Yerneni, "Template-Based Wrappers in the TSIMMIS System," In *Proceedings of Twenty-Third ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[10] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina, "Object Fusion in Mediator Systems," In *Proceedings of Twentieth International Conference on Very Large Databases*, Bombay, India, 1996

[11] S. C. Johnson, "Yacc—yet another compiler compiler," AT&T Bell Laboratories, Murray Hill, N.J., Computing Science Technical Report 32, 1975.

[12] N. Ashish and C. Knoblock. "Wrapper Generation for Semi-structured Internet Sources." *Workshop on Management of Semistructured Data*, Ventana Canyon Resort, Tucson, Arizona.

[13] L. Wall and R. L. Schwartz (1992). *Programming perl*, O'Reilly & Associates, Inc., Sebastopol, CA.

[14] Corporation for National Research Initiatives. "The Python Language Home Page." URL, http://www.python.org/, Reston, Virginia.

MedMaker: A Mediation System Based on Declarative Specifications*

Yannis Papakonstantinou, Hector Garcia-Molina, Jeffrey Ullman
Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA

Abstract

Mediators are used for integration of heterogeneous information sources. In this paper we present a system for declaratively specifying mediators. It is targeted for integration of sources with unstructured or semi-structured data and/or sources with changing schemas. In the paper we illustrate the main features of the Mediator Specification Language (MSL), *show how they facilitate integration, and describe the implementation of the system that interprets the MSL specifications.*

1 Introduction

Many applications require integrated access to heterogeneous information, stored at sources with different data models and access mechanisms [LMR90, Gup89, C+94, A+91]. The TSIMMIS data-integration system provides integrated access via an architecture (see Figure 1.1) that is common in many projects: *Wrappers* [C+94, FK93] (also called *translators* [PGMW95]) convert data from each source into a common model, as illustrated in Figure 1.1. The wrappers also provide a common query language for extracting information. Applications can access data directly through wrappers, but they may also go through *mediators* [PGMW95, Wie92]. A mediator combines, integrates, or refines data from wrappers, providing applications with a "cleaner" view. For example, a mediator for Computer Science publications could provide access to a set of bibliographic sources that contain relevant materials. Users accessing the mediator would see a single collection of materials, with, for example, duplicates removed and inconsistencies resolved (e.g., all authors names would be in the format last name, first name).

Our focus on this paper is on integration of sources that do not have a well defined static schema. This class of sources includes databases that have an often changing schema, as well as information sources that contain unstructured or semistructured data. There are many applications that use such data. A typical example is electronic mail where objects have some well defined "fields" such as the destination and source addresses, but there are others that vary from one mailer to another. Furthermore, fields are constantly being added or modified. The same situation arises with medical records, bibliographic information, knowledge representation frames[G+92], and many others.

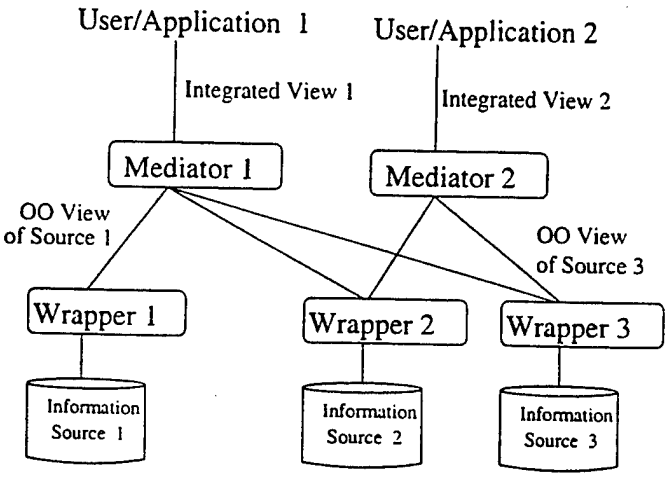


Figure 1.1: The TSIMMIS architecture for integration

1.1 The OEM Model

Most applications that have to deal with unpredictable, unstructured information use a *self-describing* model [MR87], where each data item has an associated descriptive label. In [PGMW95] we have defined a self-describing data model, called the Object Exchange Model (OEM), that captures the essential features of the models used in practice. It also generalizes them to allow arbitrary nesting and to include object identity.

To illustrate the OEM model, consider the following objects (one object per line):

```
<&1, person, set, {&11, &12, &13, &14}>
     <&11, name, string, 'George Jones'>
     <&12, department, string, 'CS'>
     <&13, relation, string, 'employee'>
     <&14,affiliations, set, {&141, &142}>
          <&141, affiliation, string, 'AI'>
          <&142, affiliation, string, 'DB'>
```

Each OEM object consists of an *object-id* (e.g., `&12`), a *label* that explains its meaning (e.g., `department`), a *type* (e.g., `string`), and a *value* of the specified type

```
<&e1, employee, set, {&f1,&l1,&t1,&rep1}>
  <&f1, first_name, string, 'Joe'>
  <&l1, last_name, string, 'Chung'>
  <&t1, title, string, 'professor'>
  <&rep1, reports_to, string, 'John Hennessy'>
<&e2, employee, set, {&f2,&l2,&t2}>
  <&f2, first_name, string, 'John'>
  <&l2, last_name, string, 'Hennessy'>
  <&t2, title, string, 'chairman'>
⋮
<&s3, student, set, {&f3,&l3,&y3}>
  <&f3, first_name, string, 'Pierre'>
  <&l3, last_name, string, 'Huyn'>
  <&y3, year, integer, 3>
⋮
```

Figure 2.2: The OEM object structure of the cs wrapper

```
<&p1, person, set, {&n1,&d1,&rel1,&elm1}>
  <&n1, name, string, 'Joe Chung'>
  <&d1, dept, string, 'CS'>
  <&rel1, relation, string, 'employee'>
  <&elm1, e_mail, string, 'chung@cs'>
<&p2, person, set, {&n2,&d2,&rel2}>
  <&n2, name, string, 'Nick Naive'>
  <&d2, dept, string, 'CS'>
  <&rel2, relation, string, 'student'>
  <&y2, year, integer, 3>
⋮
```

Figure 2.3: The OEM object structure of whois

Problems in Mediator Specification Creating the integrated view from the wrapper views requires the resolution of a number of problems:

- *schema-domain mismatch*: The whois source represents names by a long string that contains both the first and the last name, while the cs database represents names using the "last_name" and "first_name" subobjects.
- *schematic discrepancy*: Data in one database correspond to metadata of the other. In particular, the status of a person – employee or student – appears as a value in whois (it was part of a relational table), while it appears in the schema of cs (it was part of the relational schema).
- *schema evolution*: The format and contents of the sources may change over time, often without notification to the mediator implementor. For example, an attribute "birthday" may appear in either of the two sources, or the "e_mail" attribute may be dropped. We would like our mediator specification to be insensitive to as many of these changes as possible. For example, if "birthday" is included or dropped, it should be automatically included or dropped from the med view, without need to change the mediator specification.

```
<&cp1, cs_person, {&mn1,&mrel1,&t1,&rep1,&elm1}>
  <&mn1, name, string, 'Joe Chung'>
  <&mrel1, rel, string, 'employee'>
  <&t1, title, string, 'professor'>
  <&rep1, reports_to, string, 'John Hennessy'>
  <&elm1, e_mail, string, 'chung@cs'>
```

Figure 2.4: Object exported by med

- *structure irregularities:* Source whois does not follow a regular schema (i.e., it is a semistructured source.)

The Mediator Specification of med The following MSL specification MS1 defines the med mediator we have described, resolving the integration problems we have discussed above. We will explain this specification in the paragraphs that follow.

```
(MS1) Rules:
  <cs_person {<name N> <rel R> Rest1 Rest2}>
   :- <person {<name N> <dept 'CS'>
               <relation R>|Rest1}>@whois
      AND decomp(N, LN, FN)
      AND <R {<first_name FN>
              <last_name LN> | Rest2}>@cs
  External:
  decomp(string,string,string)(bound,free,free)
      impl by name_to_lnfn
  decomp(string,string,string)(free,bound,bound)
      impl by lnfn_to_name
```

A specification consists of rules that define the view provided by the mediator, and declarations of functions that will be called upon for translating objects from one format to another. Each rule (the above specification has only one rule) consists of a *head* and a *tail* that are separated by the :- symbol. The tail describes the patterns of objects that must be found at the sources, while the head describes the pattern of the top-level objects of the integrated view.

Intuitively, we may think of the process of "creating" the virtual objects of the mediator as pattern matching. First, we match the patterns that appear in the tail against the object structure of cs and whois, trying to bind the *variables* (represented by identifiers starting with a capital letter, such as N, Rest1, etc.) to object components of cs and whois. Then we use the bindings to "construct" the objects specified in the head of the rule.

The specification is based on patterns of the form <*object-id label type value*>, where we may place constants or variables in each position. For simplicity we can drop some of the fields when they are irrelevant. If one field is dropped, we assume it is the type, so we have a pattern of the form <*object-id label value*>. If two fields are dropped, we assume they are the type and the object-id. When the object-id is missing in a tail pattern, it means that we do not care about the object-id's appearing at the sources. When an object-id is missing from a head pattern, it means we do not care what object-id the mediator uses for the "generated" object.

(e.g., 'CS'). Object-ids can be of different types, but for now, think of them as arbitrary strings that are used to link objects to their subobjects. (For more details, see [PGM].) Labels are strings that are meaningful to the application or end user. Labels may have different meanings at different sources. Indeed, it will be the job of mediators to resolve these conflicts. Values may be either of an atomic type (e.g., 'George Jones' is of type string), or be a set of subobjects (e.g., the value of the "affiliations" object is {&141, &142}).

Some OEM objects (e.g., the object identified by &1) are *top-level* objects, and we write them with the leftmost indentation. For performance reasons clients query object structures starting, by default, from the top-level objects. For example, a simple query may ask for top-level "person" objects that have a "department" subobject with value 'CS'. Nevertheless, the client is not restricted to query the object structure starting from top-level objects, as will be explained in Section 2.

The OEM model forces no regularity on data. For example, a second person object may or may not have subobjects with the same labels as the person shown above. The fact that there is no schema, or each object has its own schema if you will, makes it possible to represent heterogeneous, changing information. It also facilitates the representation of information integrated from multiple heterogeneous sources, that typically have different schemas.

1.2 The Mediator Specification Language (MSL)

Given a set of sources with wrappers that export OEM objects, we would like to build mediators to integrate and refine the information. In particular, we restrict our attention to mediators that provide integrated OEM *views* of the underlying information. The significant programming effort involved in the hard-coded development of TSIMMIS mediators suggests the need for development of systems that facilitate mediator development. Our mediation system, *MedMaker*, provides a high level language, called *Mediator Specification Language (MSL)* that allows the declarative specification of mediators. At run time, when the mediator receives a request for information, MedMaker's *Mediator Specification Interpreter (MSI)* collects and integrates the necessary information from the sources, according to the specification. The process is analogous to expanding a query against a conventional relational database view. Indeed, MSL can be seen a view definition language that is targeted to the OEM data model and the functionality needed for integrating heterogeneous sources. The special requirements of integration led to the introduction of a number of useful concepts and properties, that are not found in conventional view definition languages. In this paper we present the following features:

- MSL mediator specifications can handle some schema evolution of the underlying sources without a need for rewriting of the specification.
- MSL can handle structure irregularities of the sources without producing erroneous or unexpected results.
- MSL can integrate sources for which we do not fully know their object structures.
- MSL can manipulate both the values and the descriptive semantic labels in the same fashion, getting around problems such as schematic discrepancies [KLK91].

The above capabilities are "packaged" in a high-level declarative language that combines power with simplicity and conciseness, thus allowing the client of an heterogeneous system to easily define an integrated view.

In the next section we present an extended example that illustrates the MSL language and some of its integration capabilities. Then, in Section 3 we discuss the architecture and implementation of MedMaker. Section 4 compares MedMaker to other systems for the integration of heterogeneous information sources and discusses ongoing and future work on MedMaker. The complete syntax and semantics of MSL are provided in [PGM].

2 A Mediator Specification Example

For our extended example, we consider two sources that contain information on the staff of a Computer Science department. The first source is a relational database containing two tables with schemas
employee(first_name,last_name,title,reports_to)
student(first_name,last_name,year)
A wrapper, named cs, exports this information as a set of OEM objects, some of which are shown in Figure 2.2. Notice how the schema information has now been incorporated into the individual OEM objects.[1]

A second source is a university "whois" facility that contains information about employees and students. A wrapper whois provides access to this source; several sample objects are shown in Figure 2.3. Notice that in this case there can be irregularities. For instance, object &p1 contains an email subobject while &p2 does not.

Let us now consider a mediator, called med, that has access to wrappers cs and whois and exports a set of "cs_person" objects. Our goal in this example is that each "cs_person" object represents a person appearing in both sources. The subobjects of each "cs_person" object should represent the combined information about this person. For example, since an object with information about Joe Chung is exported from both cs and whois, med combines this information and exports the object of Figure 2.4.

[1] Two minor points: (1) After translation, we have lost knowledge that objects at this source *must* have a regular structure. If this information is important to the applications, it could be exported as additional facts about this source. (2) One could consider it inefficient to repeat the schema in all objects, in this case where there is a regular pattern to objects. This problem can easily be addressed by data compression when objects are exported. Conceptually, we believe it is easier to think of each object as having its own labels.

When the *label* (*value*) field contains a constant the pattern matches successfully only with OEM objects that have the same constant in their *label* (*value*) field. On the other hand, when the *label* (*value*) field contains a variable, the pattern can successfully match with any OEM object, regardless of the label (value) of the object. For example, the pattern `<name N>` can match with OEM objects `<&1, name, string, 'Fred'>` or `<&2, name, string, 'Tom'>`. As a result of a successful matching, the variable `N` will bind to the value of the specific OEM object (either `'Fred'` or `'Tom'` in the example).

Returning to our mediator specification example, we match the patterns of the tail against the top-level objects of the corresponding sources, trying to bind the variables of the tail to appropriate object components. In particular, we match the pattern `<person {<name N> ... | Rest1}>` against the objects of source `whois`, trying to bind the variables `N`, `R`, and `Rest1` to appropriate object components. That is, we try to find top-level "person" objects that have a "name" subobject, a "dept" subobject with value `'CS'`, and a "relation" subobject. The object identified by `&p1` (see Figure 2.3) satisfies these requirements. As a result, `N` binds to `'Joe Chung'`, `R` binds to `'employee'`, and `Rest1` binds to the remaining subobjects, i.e., it binds to `{<&elm1, e_mail, string, 'chung@cs'>}` Let us name this set of bindings $b_{w,1}$. Other objects may also satisfy these conditions and produce other bindings for `N`, `R`, and `Rest1`. For instance, `N` can bind to `'Nick Naive'`, `R` to `'CS'`, and `Rest1` to `{<&y2, year, integer, 3>}`.

The specification also indicates that we match the pattern `<R {<first_name FN> ... | Rest2}>` against the objects at source `cs`, obtaining bindings for the variables `R`, `FN`, `LN`, and `Rest2`. Referring to Figure 2.2, we see that one of these binding, call it $b_{c,1}$, will bind `R` to `'employee'`, `FN` to `'Joe'`, `LN` to `'Chung'`, and `Rest2` to `{<&t1, title, string, 'professor'> <&rep1, reports_to, string, 'John Hennessy'>}`.

The next step is to match the two sets of bindings. A binding $b_{w,i}$ from `whois` matches a binding $b_{c,i}$ from `cs` if the two bindings agree on the values assigned to common variables (in this case, `R`) and the name `N` found in `whois` "corresponds" to the last name, first name pair `LN`, `FN` found in `cs`. For example, binding $b_{w,1}$ matches $b_{c,1}$ because they both bind `R` to `'employee'` and the name `N = 'Joe Chung'` corresponds to last name `LN = 'Chung'` and first name `FN = 'Joe'`.

External Predicates The correspondence between names and first, last name pairs is given by the predicate `decomp(N,LN,FN)`. Conceptually, we can think of `decomp` as a predicate that evaluates to true if `N` is a valid decomposition of last, first names `LN`, `FN`. In practice, `decomp` is implemented as a pair of functions, `name_to_lnfn` and `lnfn_to_name` (in principle written in any programming language), and defined in the mediator specification. For example, the line `decomp ... by name_to_lnfn` indicates that `name_to_lnfn` can be called with a full name (the first bound parameter); the function decomposes the name and returns the last and first names (second and third free parameters). Similarly, `lnfn_to_name` can compose a last, first name pair and produce a full name. Thus, operationally, to check if `decomp('Joe Chung', 'Chung', 'Joe')` is true, we can call `name_to_lnfn` with input parameter `'Joe Chung'` and see if it returns `'Joe'` and `'Chung'`. If it does, the predicate holds. Equivalently, we can call `lnfn_to_name` to perform the check.[2] We assume that the resulting result would be the same in any scenario. (Having more than one function for `decomp` gives flexibility at execution time.)

Creation of the Virtual Objects For each set of matching bindings from the tail patterns, we conceptually create an object in the `med` view.[3] (We stress that objects are not really materialized by the mediator specification.) The head of the rule tells us how to construct the view objects. For example, the matching bindings $b_{w,1}$ and $b_{c,1}$ result in the object of Figure 2.4.

Note that even though `Rest1` and `Rest2` are bound to sets of objects, and `<name N>` and `<rel R>` are bound to single objects, we can include all four inside the curly braces that define the subobjects for a "cs_person" object. In general, when variables that have been bound to sets appear inside curly braces {} in a rule head, the first level of their contents is "flattened out" and included in the set value that is described by the curly braces pattern.

Note also that our sample head did not specify any types or object-id's for the view objects. The types, of course, are simply set to the types of the bound variables (`string` in our case.) For the object-id's, any arbitrary unique strings can be used (e.g., `&cp1`, `&mn1`, ... are used in Figure 2.4.)

MSL's Solutions to Mediator Specification Problems The specification of `med` solves the integration problems mentioned earlier, mainly by exploiting the free use of variables in the Mediator Specification Language, and the schema/data combination ability of OEM. For example, we were able simultaneously to bind variable `R` to a value in `whois` and a label in `cs`, thus addressing the schematic discrepancy. The schema evolution problem is handled by the use of variables `Rest1` and `Rest2`. If, say, new attributes such as "birthday" are added to `cs`, no change is required to the mediator specification. The new attribute will be included with `Rest1` and propagated to the integrated view. On the same time, the bindings of variables `Rest1` or `Rest2` are not required to

[2] Of course, if the implementor had provided a function `check_name_lnfn` that is called with all three parameters bound, we would simply call `check_name_lnfn` with input parameters `'Joe Chung'`, `'Chung'`, and `'Joe'`.

[3] In reality, we first project the bindings of the variables of the tail, into bindings of the variables that appear in the head of the rule. Then we eliminate *duplicated* bindings, and finally we create an object of `med` for each set of bindings of the variables of the head.

carry homogeneous sets of objects. For example, binding $b_{w,1}$ binds `Rest1` to {<&elm1, e_mail, string, 'chung@cs'>} while $b_{w,2}$ binds `Rest1` to {}. In this way, MSL can handle the integration of unstructured sources that do not have a regular schema. Finally, the ability to use external predicates allows us to process atomic values in any desirable way.

One apparent limitation of the integrated view we have defined for `med` is that it only includes information for people that appear in both `cs` and `whois`. In particular, we may wish to include information in `med` even if it appears in a single source. In Section ?? we briefly present other features of the MSL that let us define such views and let us perform other useful integration tasks. (A complete description appears in [PGM].) However, before doing so, in the following section we illustrate how our Mediator Specification Interpreter (MSI) would process an incoming query against the sample definition we have given.

Other Features of the Mediator Specification Language In the previous two sections we illustrated the basic functionality of the MSL language. The language has additional features specifically designed to facilitate the integration and querying of heterogeneous sources. Due to space limitations, we cannot provide examples for all these features. Instead, we briefly summarize some features and refer the reader to [PGM] for examples and details.

First, note MSL's ability to *retrieve schema information*: One can place variables in the label positions of an MSL query, and thus retrieve information about labels and the object structure of a source. This is a useful feature for exploring new or changing sources. Second, MSL provides the *wildcard* feature that allows searches for objects at any level in the object structure of the source, without need to specify the entire path to the desired object. The wildcard feature is especially useful when we form queries without complete knowledge of the structure of the underlying data. (Without appropriate index structures, wildcard searches may be expensive, so some sources may not support them or may support them in a restricted fashion.) Finally, MSL allows the specification of *semantic object-id's* that semantically identify an exported object and they have meaning beyond the mediator call that yielded them. Semantic object-id's provide a powerful mechanism for object fusion. Due to space limitations we will not discuss any further this feature.

3 Architecture and Implementation of MSI

The Mediator Specification Interpreter (the runtime component of MedMaker) processes a query using a pipeline with the following three components (see Figure 2.5):

1. The *View Expander and Algebraic Optimizer (VE&AO)* reads the query and the mediator specification and discovers which objects it must obtain from each source. Furthermore, it determines the conditions that the obtained source objects must satisfy.

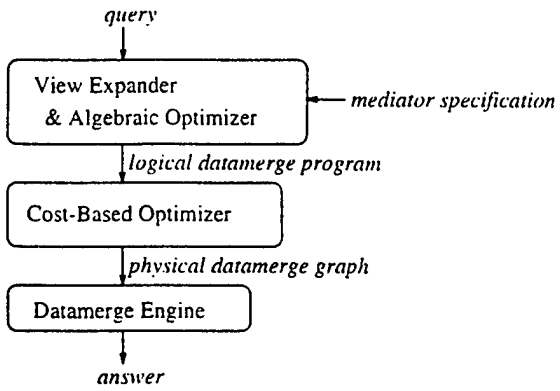


Figure 2.5: The basic architecture of MSI

2. The *cost-based optimizer* develops a plan for obtaining and combining the objects specified by the VE&AO. The plan specifies what queries will be sent to the sources, in what order they will be sent, and how the results of the queries will be combined in order to derive the result objects.

3. The *datamerge engine* executes the plan and produces the required result objects.

In the following subsection we use an example to overview MSI's query processing. Subsections 3.2 to 3.5 discuss each component, the languages at each interface, and various interesting query decomposition and optimization issues.

3.1 Query Processing Overview

Let us assume that a client of mediator `med` wants to retrieve all the data for 'Joe Chung.' In this paper, we use MSL (with one minor modification discussed below) as our query language.[4] The use of MSL simplifies our discussion, and furthermore, MSL makes a good query language because of its power and simplicity. Using MSL, our query can be expressed as:

(Q1) `JC :- JC:<cs_person {<name 'Joe Chung'>}>@med`

The object pattern (or object patterns) that appears in the tail of the query are matched against the object structure of `med` in exactly the same manner that tail patterns of MSL rules are matched against the sources. One new MSL feature that appears in the tail of our sample query is the *object variable* `JC`. The operator : indicates that `JC` must bind to "cs_person" objects that have a "name" subobject with value 'Joe Chung'. The query head indicates that every object

[4]The TSIMMIS project at Stanford is also exploring a different query language, called *LOREL*. It is an object-oriented extension to SQL and is oriented to the end-user. LOREL is described in [Q+]. MSL is more powerful than LOREL (e.g., MSL allows the specification of recursive views) and is targeted to mediator specification.

that JC binds to is included in the result. Unlike mediator specification, when MSL is used for querying, the objects specified by the query rule head are materialized at the client.[5]

View Expansion Given our sample query, the VE&AO replaces the object pattern of the query tail with object patterns that refer to objects of the sources, thus deriving the datamerge rule R2:

```
(R2) <cs_person {<name 'Joe Chung'> <rel R>
                 Rest1 Rest2}>
     :- <person {<name 'Joe Chung'> <dept 'CS'>
                 <relation R> | Rest1}>@whois
        AND decomp('Joe Chung', LN, FN)
        AND <R {<first_name FN> <last_name LN>
                | Rest2}>@cs
```

Intuitively, the MSI derived the above rule by matching the pattern JC:<cs_person ...>@med of the query tail against the head of the mediator specification rule of med.[6] After the matching, we generate a datamerge rule whose head is the head of the query and whose tail is the mediator specification rule's tail.

Execution Plan Now that the MSI knows what objects it has to find at the sources, the cost-based optimizer builds a physical datamerge program that specifies what queries should be sent to the sources, in what order they should be sent, and how the results of the queries should be combined in order to produce the query result. Here we informally describe a possible (and efficient) plan for our running example:

1. Bindings for the variables R and Rest1 are obtained from the source whois. The bindings are obtained in two steps. First the following query is sent to whois:

   ```
   <bind_for_whois {<bind_for_R R>
                    <bind_for_Rest1 Rest1>}>
    :- <person {<name 'Joe Chung'> <dept 'CS'>
                <relation R> | Rest1}>@whois
   ```

   Labels bind_for_whois, bind_for_R and bind_for_Rest1 are simply place-holders that allow the MSI to conveniently pick out the desired information from the returned result objects.

2. Bindings for LN and FN can be obtained from one of the decomp functions, i.e., from name_to_lnfn. We call it with bound parameter N = 'Joe Chung' and obtain LN = 'Chung' and FN = 'Joe'.

3. For each of the R binding of step (1), we combine it with the single binding of step (2), and submit a query to cs to obtain a binding for Rest2. For example, for the binding R = 'employee' we send the following query to cs:

   ```
   <bind_for_cs {<bind_for_Rest2 Rest2>}>
    :-<employee {<first_name 'Joe'>
                 <last_name 'Chung'>|Rest2}>@cs
   ```

4. Once MSI obtains bindings for Rest2 as well, it generates objects that follow the pattern of the head of (R2). For example, considering the bindings we have illustrated so far, the MSI would generate the object of Figure 2.4.

3.2 View Expansion and Algebraic Optimization

The VE&AO matches the query against the mediator specification rules and rewrites the query so that references to the virtual mediator objects are replaced by references to source objects. The result is a *logical datamerge program* that is a set of MSL rules specifying the result. In Section 3.1 we illustrated the view expansion and algebraic optimization process. There, expression (R2) was the logical datamerge program. In the rest of this section we explain the VE&AO process in more detail. In general, the VE&AO formulates the logical datamerge programs in two steps:

- First it matches the query tail conditions with rule heads. The successful matches result in expressions called *unifiers*. Intuitively, our unifiers describe the match between the query and the rule, the conditions that must be pushed to the sources, and other information necessary for the rewriting of the query. Note, they can be viewed as extensions of the unifiers used in resolution of first order clauses [GN88].

- Then for every unifier a logical datamerge rule is formed. The rule head is formed by applying the unifier to the query head, while the rule tail is formed by applying the unifier to the mediator specification rule tails and subsequently forming their conjunction.

For example, consider the query Q1 (Section 3.1) and the specification MS1 of med. The match[7] results in the unifier θ_1 where

$$\theta_1 = \left[\begin{array}{l} \texttt{N} \mapsto \texttt{'Joe Chung'}, \\ \texttt{JC} \Rightarrow \quad \texttt{<cs\_person \{<name 'Joe Chung'>} \\ \qquad\qquad \texttt{<rel R> Rest1 Rest2}\texttt{\}>} \end{array} \right]$$

The above unifier consists of one *mapping*, indicated by the $\mapsto$, and one *definition*, indicated by the $\Rightarrow$. (The need for discriminating between mappings and definitions will become apparent in the next paragraphs.) The application of θ_1 to the query head

[5] Here we do not address the problem of materializing OEM objects at clients. The various issues and strategies are discussed in [PGMW95].

[6] If there were several rules, the MSI would look for one or more matching rule heads. If more than one head matches, then more than one rule will be considered; resulting objects will be added to the result.

[7] Before we match a query with one or more rules we must rename the variables that appear in the query and the rules, so that no two rules, or a query and a rule have identically named variables.

causes the substitution of JC by the structure following the $\Rightarrow$. Similarly, the application of θ_1 to the mediator rule tail causes the substitution of N by 'Joe Chung'. Combining the transformed query head with the transformed mediator rule tail we obtain the logical datamerge rule Q2.

In general, a unifier may contain any number of mappings and/or definitions. When the VE&AO matches a query condition with a rule head it generates all unifiers θ such that

1. If we apply the mappings to the query condition and the mediator rule head, the transformed query condition pattern is *contained* in the rule head pattern. In the example, the transformed query condition <cs_person {<name 'Joe Chung'>}> is contained in the transformed rule head <cs_person {<name 'Joe Chung'> <relation R> Rest1 Rest2}> because they have the same label cs_person and every subobject pattern of the query condition pattern (i.e., the pattern <name 'Joe Chung'>) is identical to a subobject pattern of the rule head. Containment guarantees that any mediator object generated by the transformed rule satisfies the query condition pattern.

2. There is a definition for every object, value, or "rest" variable that appears in the query head and also appears in the query tail preceding a ":". The definition carries all the information about the structure of the mediator objects that bind to the query variable. For example the definition of JC carries all the required information about the mediators cs_person objects.

3.3 Pushing Conditions to the Sources

The VE&AO pushes conditions such as "the name must equal 'Joe Chung'" to the corresponding source. Indeed, VE&AO pushes to the sources all conditions that can be pushed, thus implementing the (well-known in relational DB's) "push selections down" algebraic optimization. In our environment with nested objects that may have unknown structure, algebraic optimization is substantially more challenging than in a relational environment. To illustrate this point, assume that the following query, that retrieves the data of 3rd year students, is sent to mediator med (specified by MS1):

```
S :- S:<cs_person {<year 3>}>@med
```

Mediator med joins data from two sources, and we cannot tell in advance whether the "year" object comes from one source or the other. In particular, when we match the query against the mediator specification, the <year 3> pattern can be "pushed" either into Rest1 or into Rest2. The two possibilities correspond to the unifiers τ_1 and τ_2:

$$\tau_1 = \left[\begin{array}{l} \texttt{Rest1} \mapsto \{< \texttt{year 3} >\}, \\ \texttt{S} \Rightarrow < \texttt{cs\_person} \quad \{< \texttt{name N} > < \texttt{rel R} > \texttt{Rest1 Rest2}\} > \end{array}\right]$$

$$\tau_2 = \left[\begin{array}{l} \texttt{Rest2} \mapsto \{< \texttt{year 3} >\}, \\ \texttt{S} \Rightarrow < \texttt{cs\_person} \quad \{< \texttt{name N} > < \texttt{rel R} > \texttt{Rest1 Rest2}\} > \end{array}\right]$$

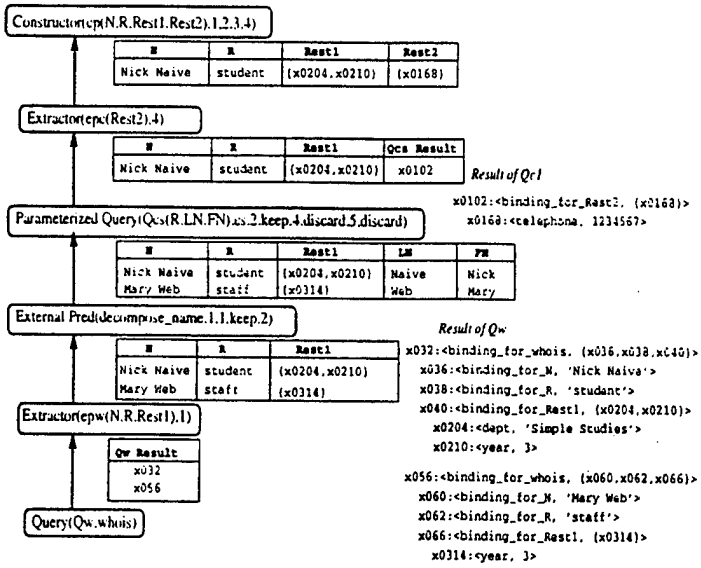


Figure 3.6: A physical datamerge graph

The two unifiers give rise to the following two rules, that constitute the logical datamerge program:

```
(R3)<cs_person {<name N> <rel R> Rest1 Rest2}>
     :- <person {<name N> <dept 'CS'> <relation R>
                 |Rest1:{<year 3>}}>@whois
        AND decomp(N, LN, FN)
        AND <R {<first_name FN> <last_name LN>
                |Rest2}>@cs
(R4)<cs_person {<name N> <relation R> Rest1 Rest2}>
     :- <person {<name N> <dept 'CS'> <relation R>
                 | Rest1}>@whois
        AND decomp(N, LN, FN)
        AND <R {<first_name FN> <last_name LN>
                | Rest2:{<year 3>}}>@cs
```

Note, mappings of the form Rest1 $\mapsto$ {< year 3 >} cause the attachment of the conditions specified inside the {} to the specified variable (Rest1 in the example). If Rest1 has already some conditions S associated with it, VE&AO would merge S with the the <year 3> condition.

Note, the examples of the previous paragraphs dealt with single condition queries. Nevertheless, the demonstrated techniques have been easily extended and implemented for multiple condition queries.

3.4 The Physical Datamerge Graph and the Datamerge Engine

The optimizer receives the logical datamerge program from the VE&AO and generates a *physical datamerge graph*. This graph specifies the queries to be sent to the sources as well as the mechanics for constructing the query result from the results received from the sources. The graph is then executed by the datamerge engine, which produces the query result.

In this section we illustrate datamerge graph execution through a detailed example. Our goal is not to describe our implementation in full detail, but rather to show the capabilities of our datamerge engine. As our

starting point we use logical datamerge rule Q3. From it, the optimizer may generate the physical datamerge graph of Figure 3.6. This is a "dataflow" graph, where the nodes (rounded boxes) represent the operations to be executed by the engine. The rectangles next to the arcs of the graph represent tables that flow during a sample run of this graph. Typically, the tuples of the tables carry bindings for the logical datamerge program variables.

The datamerge engine executes the graph in a bottom-up fashion. First, the lower *query* node is executed. This causes query Qw to be sent to source whois, obtaining bindings for N, R, and Rest1. Query Qw is provided to the engine by the optimizer, and is defined as:

```
(Qw) <bind_for_whois {<bind_for_N N>
                      <bind_for_R R>
                      <bind_for_Rest1 Rest1>}>
     :- <person {<name N> <dept 'CS'>
                 <relation R>
                 | Rest1:{year 3}}>@whois
```

The result of Qw is placed in the mediator's memory. In Figure 3.6 we show this result at the bottom of the figure. The numbers with a "x" prefix represent object addresses in the mediator's memory. For example, one result object is at address x032; it has label bind_for_whois and its value is a set containing the objects at locations x036, x038 and x040. For readability, we omit the object-id and type fields of the objects from the figure.

The query operator produces a table where each line contains the address of a top-level result object (x032 and x056 in the example). For readability, we add a heading row to our tables (Qw Result in this case), but these do not appear in practice.

The table is passed to the next operator in the graph, an *extractor* node that extracts bindings of the variables N, R, and Rest1 (from the "binding_for_whois" objects) and outputs a table of corresponding (N, R, Rest1) tuples. The extractor node has two parameters: the first is the optimizer provided object pattern epw, defined by

```
<bind_for_whois {<bind_for_N N> <bind_for_R R>
                 <bind_for_Rest1 Rest1>}>
```

epw indicates where the desired bindings are found in the result objects; the second parameter (1) indicates the column of the input table that contains the objects that are the subject of the extraction. Again, the heading row in the output table is only for readability. Also for reabability, in the N and R columns we write strings, while in reality we have pointers to the strings. Similarly, in the Rest1 column we write the full sets while in reality the column contains pointers to the indicated sets.

Then, for every tuple, the *external pred(-icate)* node invokes the predicate decomp. The other parameters for this node indicate: the number of arguments for decomp (1); the column of the input table containing the one input parameter (1); whether the input column is kept in the output table;[8] and the number of result arguments from decomp (2).

The next node is the *parameterized query* node. For each tuple of its input table, this node generates a query for source cs requesting bindings for Rest2 that are needed to construct final result objects. The query to send is defined by Qcs which is provided by the optimizer along with the graph:

```
(Qcs(R,LN,FN)) <bind_for_Rest2 Rest2>
                :-<$R {<last_name $LN>
                       <first_name $FN>|Rest2}>@cs
```

The values for query parameters $R, $LN, and $FN are taken from the 2nd, 4th, and 5th columns of the incoming table. (The keep and discard parameters again indicate if the inputs columns remain in the output table.) Thus, for our sample data, two queries Qcs1 and Qcs2 are emitted:

```
(Qc1) <bind_for_Rest2 Rest2>
      :-<student {<last_name 'Naive'>
                  <first_name 'Nick'>|Rest2}>@cs
(Qc2) <bind_for_Rest2 Rest2>
      :-<staff {<last_name 'Web'>
                <first_name 'Mary'>|Rest2}>@cs
```

Let us assume that Qc1 returns only the x0102 "binding_for_Rest2" object and Qc2 does not return anything. In this case, the parameterized query node outputs the table shown in Figure 3.6. After the upper extractor node extracts Rest2 bindings from the results of the parameterized query node, the *constructor node* is activated and creates the final result objects. The form of these objects is defined by the pattern cp(N,R,Rest1,Rest2) where

```
cp(N,R,Rest1,Rest2) =
 <cs_person {<name N> <relation R> Rest1 Rest2}.
```

For each row in the input table, the constructor operator takes a row (1st, 2nd, 3rd, and 4th values), assigns them to the N, R, Rest1, and Rest2 values in cp, creating one of the final result objects.[9]

Through this example we have illustrated how the entire mediation process can be described by a low level executable graph. The nodes of our datamerge graphs are the "machine language" of MedMaker which is run by our implementation of the datamerge engine. (Indeed, it is interesting to compare them with relational algebra expressions.)

3.5 Cost-Based Optimization Challenges

There are more than one physical datamerge graphs that correspond to a logical datamerge program. The optimizer has to select the "optimal" graph. However, optimization of a powerful object-oriented language that operates on autonomous and heterogeneous information sources is much harder than the optimization of traditional SQL queries on a conventional database. Our current implementation uses some very simple heuristics to guide datamerge graph selection. This seems to work well, at least for simple scenarios. In the rest of this section we briefly discuss some of our research directions for optimization.

[8] As opposed to extractor nodes that always dicard their input column (after using it).

[9] Our current implementation does not have a duplicate elimination feature, though the MSL semantics describe duplicate elimination in the OEM context.

Note the following two hard problems for the cost-based optimizer of a mediator: First, the limited query capabilities of the underlying sources may prohibit even simple algebraic optimizations, such as "push selections and projections down". For example, the source `whois` may not be able to evaluate the condition on "year" that appears in `Qw`. A solution to this problem appears in [PGH]. A second problem arises when the wrappers do not provide cost and statistics information. In this case, the optimizer has to rely on ad-hoc heuristics (e.g., the outer patterns of the join order are the ones that have the greatest number of conditions) or tries to build its own statistics database that is based on results of previous queries and on sampling.

4 Related Work and Discussion

In this section we contrast MedMaker to other heterogeneous information source integration systems, we discuss our motivation behind the design of OEM and MSL, and we describe our ongoing work on MedMaker.

It is widely accepted that the relational data model and the corresponding view definition languages are insufficient to provide integration, even of relational databases [KLK91]. Thus, many projects have adopted (or defined) OO models to facilitate integration (some examples are [C+94, A+91]). We described in Sections 1 and 2 the OEM features that make it suitable for integration of heterogeneous information systems. Another difference between OEM and conventional OO models is that OEM is much simpler and does not have a strong typing system. OEM supports only *object nesting* and *object identity*, while other features, such as classes, methods, and inheritance are not supported directly. (Nevertheless, classes and methods can be "emulated" [PGMW95]).

We believe that MSL mediator specifications tend to be short and simple and avoid questions such as "what is the class of the view objects?", that complicate object-oriented view definition[AB91]. In spite of its simplicity, MSL is quite powerful. For instance, it allows the construction of arbitrarily complex object structures (which XSQL [KKS92] does not).

MSL and OEM can be seen as a form of first-order logic. Indeed, we borrow many concepts from logic oriented languages such as datalog [Ull88, Ull89], HiLog [CKW93], O-Logic [Mai86], and F-Logic [KL89]. HiLog first proposed – under a logic framework – the idea of mixing schema and data information.[10]

A very important difference between MedMaker and other integration systems is that MedMaker can integrate conventional well-structured databases that have a static schema and at the same time can integrate sources that do not have a regular schema, or sources that have an often-changing schema. The ability to integrate all kinds of sources is due to:

1. OEM's absence of schema, that allows the intuitive representation of heterogeneous, semistructured, and changing information.
2. MSL's ability to exploit regularities and complete knowledge of the schema (the example of Section 3.3 demonstrated the tradeoff between performance and partial knowledge of the schema).

Though systems that integrate well-structured conventional databases exist (e.g., [A+91, K+93, BLN86, LMR90, T+90, Gup89]) and recently systems for the integration of sources with minimal structure have also appeared [Fre, S+93], we do not know of view definition based systems ([A+91, Ber91, CWN94] and others) that handle the whole spectrum of information sources simultaneously, and with MSL's flexibility.

Note, MedMaker performs integration by "working" with the structures of the source objects. Semantic information is effectively encoded in the MSL rules that do the integration. There are many projects that follow MedMaker's "structural" approach [Ber91, DH86, B+86], as well as many projects that follow a semantic approach [HM93, H+92]. We believe that the power of the structural approach, along with the flexibility, generality, and conciseness of OEM and MSL make the "structural" approach a better candidate for the integration of widely heterogeneous and semistructured information sources.

[10] [KLK91] has also proposed an interesting mixing of schema and data information for the relational data model.

Acknowledgments

We are grateful to Koichi Munakata for designing and implementing the datamerge engine. We also thank Joachim Hammer, Pierre Huyn, Dallan Quass, Anand Rajaraman, Anthony Tomasic, Vasilis Vassalos, Jennifer Widom, and the entire Stanford Database Group for numerous fruitful discussions and comments.

References

[A+91] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference*, pages 238–47, Denver, CO, May 1991.

[B+86] Y.J. Breibart et al. Database integration in a distributed heterogeneous database system. In *Proc. 2nd Intl. IEEE Conf. on Data Engineering*, Los Angeles, CA, February 1986.

[Ber91] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. In *Proc Intl Workshop on Interoperability in Multidatabase Systems*, pages 22–29, Kyoto, Japan, 1991.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[C+94] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. Hilog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, February 1993.

[CWN94] S. Chakravarthy, Whan-Kyu Whang, and S.B. Navathe. A logic-based approach to query processing in federated databases. *Information Sciences*, 79:1–28, 1994.

[DH86] U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. In *Proc. IEEE Workshop on Object-Oriented DBMS*, Asilomar, CA, September 1986.

[FK93] J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.

[Fre] M. Freedman. WILLOW: Technical overview. Available by anonymous ftp from `ftp.cac.washington.edu` as the file `willow/Tech-Report.ps`, September 1994.

[G+92] M.R. Genesereth et al. Knowledge Interchange Format. Version 3.0. Reference Manual. Technical Report Logic-92-1, Stanford University, 1992. Also available by URL `http://logic.stanford.edu/kif.html`.

[GN88] M.R. Genesereth and N.J. Nillson. *Logical Foundations of Artificial Intelligence*. Morgan Cauffman, 1988.

[Gup89] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases*. IEEE Press, 1989.

[H+92] M. Huhns et al. Enterprise information modeling and model integration in Carnot. Technical Report Carnot-128-92, MCC, 1992.

[HM93] J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *Intl Journal of Intelligent and Cooperative information Systems*, 2:51–83, 1993.

[K+93] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD*, pages 59–68, 1992.

[KL89] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf.*, pages 134–46, Portland, OR, June 1989.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of heterogeneous databases with schematic discrepancies. In *Proc. ACM SIGMOD*, pages 40–9, Denver, CO, May 1991.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[Mai86] D. Maier. A logic for objects. In J. Minker, editor, *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, Washington, DC, USA, August 1986.

[MR87] L. Mark and N. Roussopoulos. Information interchange between self-describing databases. *IEEE Data Engineering*, 10:46–52, 1987.

[PGH] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. Available via ftp at `db.stanford.edu` file `/pub/papakonstantinou/1995/cbr-extended.ps`.

[PGM] Y. Papakonstantinou and H. Garcia-Molina. Object fusion in mediator systems (extended version). Available by anonymous ftp at `db.stanford.edu` as the file `/pub/papakonstantinou/1995/fusion-extended.ps`.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[Q+] D. Quass et al. Querying semistructured heterogeneous information. To appear in DOOD95. Available by anonymous ftp at `db.stanford.edu` as the file `/pub/quass/1994/querying-submit.ps`.

[S+93] K. Shoens et al. The Rufus system: Information organization for semistructured data. In *Proc. VLDB Conference*, Dublin, Ireland, 1993.

[T+90] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems*. Computer Science Press, New York, NY, 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*. Computer Science Press, New York, NY, 1989.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

A Toolkit for Constraint Management in Heterogeneous Information Systems*

Sudarshan S. Chawathe
Hector Garcia-Molina
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
{chaw,hector,widom}@cs.stanford.edu

Abstract

We present a framework and a toolkit to monitor and enforce distributed integrity constraints in loosely coupled heterogeneous information systems. Our framework enables and formalizes weakened notions of consistency, which are essential in such environments. Our framework is used to describe (1) interfaces provided by a database for the data items involved in inter-site constraints; (2) strategies for monitoring and enforcing such constraints; (3) guarantees regarding the level of consistency the system can provide. Our toolkit uses this framework to provide a set of configurable modules that are used to monitor and enforce constraints spanning loosely coupled heterogeneous information systems.

1 Introduction

We address the management of distributed integrity constraints over data that is stored in a collection of loosely coupled heterogeneous information systems. Distributed integrity constraints arise naturally whenever data that is semantically related is stored in different systems. For example, a construction company keeps data about a building under construction in its private database. This data must be consistent with the architect's design (e.g., walls must be in the same places), which is stored in an entirely different database.

Throughout this paper, we use the term "database" to mean any information system. In addition to traditional database systems, we include bibliographic information systems, "whois" servers, legacy systems, file systems, etc. We use the term "loosely coupled" to refer to information systems that do not offer standard control facilities such as those found in traditional distributed databases. In particular, such databases do not support multi-database transactions or multi-database query and update mechanisms that guarantee data consistency. Often, one or more of the component databases does not even support (local) transactions. Another characteristic of such

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Anderson Faculty Scholar Fund, the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

environments is that different databases support different modes of access to the database. For example, one database might provide only read access to its data, while another might provide both read and write access. Yet another may provide notification of updates.

This heterogeneity in the method of database access and control is one of the prime reasons why traditional integrity constraint management techniques cannot be applied to loosely coupled heterogeneous environments. In particular, traditional approaches to constraint management assume various facilities such as distributed transactions, remote locking, and prepare-to-commit interfaces, which are usually not supported by the databases involved in a loosely coupled system. Further, most previous work assumes that all databases in the system offer a homogeneous access and control method, as discussed in Section 2.

In spite of the difficulties outlined above, integrity constraint management is very important in many loosely coupled scenarios. Currently, systems involving data stored in several loosely coupled databases have no systematic method for monitoring or enforcing integrity constraints over the data. In most such systems, integrity constraints are simply not monitored, are monitored manually, or are monitored in an ad-hoc manner. Monitoring integrity constraints manually or using ad-hoc techniques is tedious and error-prone, while neglecting integrity constraint management altogether often leads to irreparable inconsistencies and costly correction measures.

We argue that in the loosely coupled environment that we study, it is not, in general, possible to make the kind of strict consistency guarantees that traditional constraint management systems make. For example, it is usually not possible to ensure that every application sees strictly consistent data any time it executes. Given this situation, our work focuses on how weakened notions of consistency may be defined, implemented, and used. We propose a uniform, rule-based framework in which we can formally define *guarantees* of weak consistency. Our framework is also used to define the *interfaces* (modes of access) provided by each database to the constraint manager. Further, we use our rule-based framework to specify constraint management *strategies*. We consider two kinds of constraint management. Constraint *enforcement* involves doing the work necessary to make (usually a weakened form of) the constraint valid. In some situations, however, the best we can do is constraint *monitoring*. This involves indicating when the constraint is valid and when it is not. We have also developed a set of proof rules that enable us to derive the validity of guarantees based on interface and strategy specifications [CGMW94]. However, due to space constraints, we do not discuss that work in this paper.

Using our framework, it is possible to capture a wide variety of constraint management techniques over a wide variety of loosely coupled heterogeneous environments, and to provide formal guarantees for weakened constraints. We also show how our interfaces and strategies can be used in practice. We describe a toolkit of general-purpose constraint management and translation services that can easily be configured to a given heterogeneous environment (for e.g., relational databases, object-oriented databases, file systems, bibliographic information systems etc.) Using the toolkit, one can describe the interfaces available for each database, and one can select strategies from a menu of proven strategies (examples of which are given in this paper) that conform to the inter-

faces. Based on the interfaces and the strategy, the toolkit offers guarantees of weak consistency to applications. At run-time, the toolkit will monitor or enforce the constraints so that the guarantees are valid.

This paper is organized as follows. In Section 2, we discuss how our work relates to traditional database constraint management literature as well as other related areas. We present a short overview of the framework underlying our approach to constraint management in Section 3. We expand on the framework by discussing interfaces, strategies and guarantees in Sections 3.1, 3.2, and 3.3, respectively. In Section 4, we present our constraint management toolkit and illustrate its use with an extended example. We discuss how failures are handled in Section 5. Some additional scenarios that illustrate the use of our framework and toolkit are presented in Section 6, while Section 7 discusses the use of guarantees and some issues in implementing distributed strategies. We summarize our conclusions in Section 8. The appendix presents the formal definition of the rule language used in our framework, which is described informally in Section 3.

2 Related Work

Most previous work in database constraint management addresses centralized or tightly coupled distributed environments. The techniques presented in such work are not applicable in the loosely coupled, heterogeneous environment we study because they assume many facilities such as distributed transactions, remote locking, prepare-to-commit interfaces, etc. For example, both [SV86] and [Gre93] provide useful techniques for monitoring constraints in distributed databases, but these techniques rely on a traditional notion of data fragmentation and the presence of global transactions.

Reference [CW93] describes a constraint maintenance method for a multi-database environment in which each database is relational, supports basic SQL operations, and has a production rules facility; in addition, there must be a persistent queue facility between sites. Similarly, [RSK91] describes a framework and [GW94] presents a set of protocols for inter-database constraints based on a homogeneous relational interface to each database. Neither approach is applicable in a truly heterogeneous environment where each database offers a different interface to the constraint manager, and where some (or all) of the databases may not have the required features.

There has been some work on specific constraint management strategies in a loosely coupled environment. For example, the *Demarcation Protocol* [BGM92] is a method to maintain simple arithmetic constraints. Reference [GW93] describes a method for checking distributed constraints at a single site whenever possible. These are are special cases of the more general framework we present here. (In fact, we can express the Demarcation Protocol in our framework and prove the associated guarantee. This is discussed in Section 6.)

Another approach to constraint management in multi-database environments is to extend the transaction concept to multi-databases by suitably weakening the traditional notion of correctness of schedules [Elm91]. This approach typically restricts the data items that may be involved in a

constraint (e.g., constraints may be over local data only for *local-serializability* [BGMS92]). These approaches differ from ours in that with extended transactions there still is no mechanism to allow different interfaces at the participating sites, and no way to monitor constraints that hold only at particular times.

Finally, the formal aspects of our framework are related to work in Metric Temporal Logic (MTL) [Koy92]. Our formalism can be considered an extension of MTL in which events are modeled explicitly, and distributed rules are used as the primary constructs for specification. A formalism of events and rules is more convenient than a purely state-based formalism for studying many systems like the ones we model; a similar observation is made in [L+93]. While our formal framework shares its interest in specification with software modeling languages such as LOTOS [BB87] and Esterel [BG92], our formalism is much simpler than those languages since it is targeted at modeling constraint management systems.

3 Framework

In this section, we describe our logical constraint management architecture and define the three main components of our framework, namely *interfaces*, *strategies*, and *guarantees*. (The toolkit is covered in Section 4.) In the interest of saving space and keeping the discussion intuitive, we introduce the concepts in our framework by example. In the appendix, we present the formal definition of the rule language we use, and its execution semantics. appendix. As mentioned in Section 2, the theoretical framework is similar to Metric Temporal Logic [Koy92], with some additions that make it easy and natural to specify constraint management in loosely coupled systems.

The key components of the logical architecture are illustrated in Figure 1. Our distributed constraint manager (CM) consists of a collection of Constraint Manager Shells (CM-Shells). The CM-Shell interacts with the local database and cooperates with other CM-Shells to monitor or enforce the inter-site constraints. If it is not possible to have a CM-Shell at the site of some database, its duties can be performed by one or more of the other CM-Shells, as for Site 3 in the figure. In the sequel, we use the term CM to refer to one or more of the CM-Shells acting on the behalf of the constraint manager. The three major components of our framework are described below:

- **Interfaces.** For each data item involved in a constraint, the interface for that data item describes, how the item may be read, written, and/or monitored by the CM. For example, the interface for a data item X might specify that a request from the CM to read X will be serviced within t_1 seconds,[1] and that any user update to X will result in a notification to the CM within t_2 seconds. The interface for each data item is dependent on the facilities provided by the database system containing that item. Note that we do not fix a specific granularity

[1] We consider seconds as our time unit in this paper, but our approach applies equally well with other time units. Note also that the use of time does not, in general, require synchronized clocks; this issue is discussed further in Section 7.

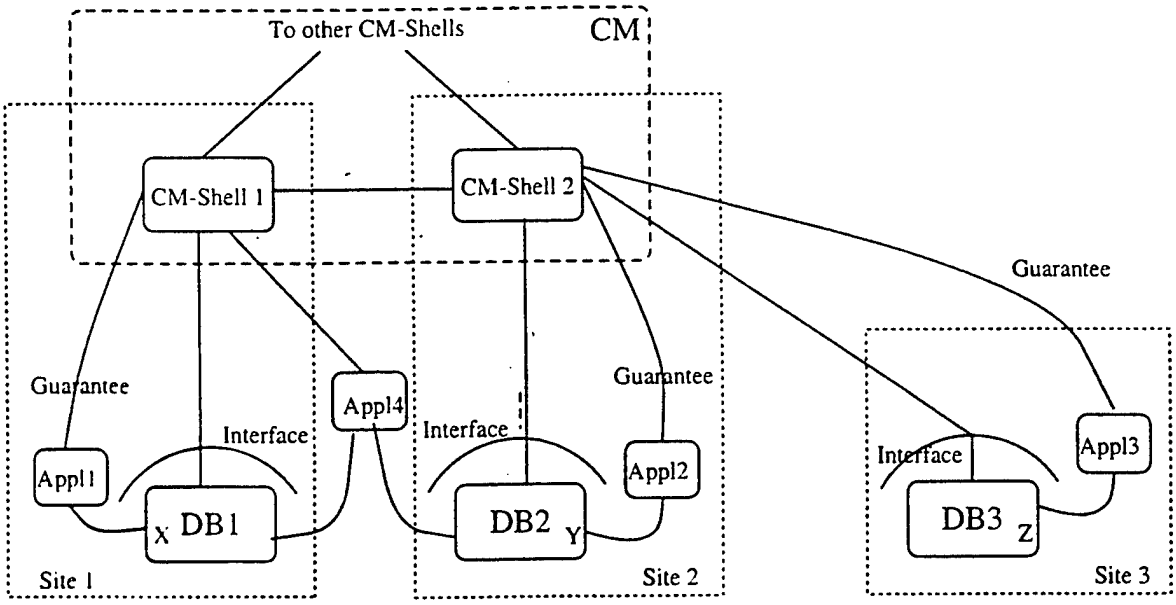


Figure 1: Constraint Management Architecture

for "data items" here. For example, a data item might be a single object or it might be the set of all tuples in a database relation. Our framework also lets us define a single interface for a set of related data items (e.g., the set of salaries of all employees in the Sales department).

- **Strategies**. For a given constraint, a strategy is the specification of an algorithm used by the CM for monitoring or enforcing the constraint. Strategies incorporate the operations available for the data items involved in the constraint. For example (informally), a naive strategy for maintaining the copy constraint $X = Y$ might specify that all updates to X are propagated to Y, while all updates to Y are undone.

- **Guarantees**. For a given constraint, a guarantee is a logical description of the level of consistency guaranteed for that constraint. For example (informally), given a *copy constraint* $X = Y$, where X is the primary copy, a guarantee may state that Y takes every value that X takes; that is, that no values of X are "lost."

Figure 1 depicts the relationship between the Constraint Manager, the databases, and the applications (or users). Each database offers interfaces to the CM for its data items. Applications inform the CM of each constraint that needs to be maintained. The CM provides guarantees to the applications, based on the interfaces and the strategy it decides to follow in order to maintain the constraint. Our approach applies to both single-site and multi-site applications. In the case of multi-site applications (for e.g., application 3 in the figure), the application chooses the CM-Shells at one of its sites to be its "local" CM-Shell. This choice is arbitrary and does not affect the validity of our approach.

Our formal framework includes detailed semantics and proof rules that allow us to prove guarantees from interface and strategy specifications. While arbitrary interfaces, strategies and guarantees may be expressed using our framework, in practice we expect most often to use interfaces and strategies from menus provided by the toolkit, with previously proven guarantees. We also plan to extend the toolkit so that it can help the system designer derive new guarantees for different interfaces and strategies.

3.1 Specifying Interfaces

The interface for a data item involved in a constraint describes how that data item may be read, written, or monitored by the constraint manager. Interfaces are specified using a rule-based notation. Note that, as in any specification system, it is important for interface specifications to correctly reflect the actual behavior provided by the database containing that item. As stated above, the database administrators at each site can choose the appropriate interfaces from a menu or they may write their own custom interfaces.

For each data item, its interface is defined by a set of *interface statements* of the form:

$$\mathcal{E}_1 \wedge C \rightarrow_\delta \mathcal{E}_2$$

The meaning of this statement is the following: If an *event* E_1, of the form indicated by *event template* $\mathcal{E}_1$, occurs at time t, and condition C (involving the event and local data items) is true at t, then the database guarantees that an event E_2 matching template $\mathcal{E}_2$ will occur at some time in the interval $[t, t+\delta]$. The condition C is evaluated at the time the left-hand side event occurs, and it may be omitted when not needed.

3.1.1 Examples

In the heterogeneous systems we model, interfaces for data items may vary within and across database systems. These interfaces can be quite varied and complex. We believe that our language is useful to describe many interfaces that occur in practice. Below, we present some examples of interfaces. We use the term CM to denote the CM-Shell responsible for the database offering the interface (usually the local CM-Shell).

Write Interface: When a database offers a write interface for a data item X, it promises to perform write operations to X requested by the CM within some time bound. We use the event template $WR(X, b)$ to represent the database receiving a request for the write operation $X \leftarrow b$ from the CM. Similarly, we use the event template $W(X, b)$ to represent the database performing the operation $X \leftarrow b$. Let δ be the time bound within which the database promises to perform the requested write operation. This write interface is expressed as follows:

$$WR(X, b) \rightarrow_\delta W(X, b)$$

Note that *parameters* such as b in the above rule are to be distinguished from local data items. Parameters are simply artifacts of the rule language, whereas local data items refer to actual data in the local database. We represent parameters by lower-case letters and local data items by upper-case letters.

No Spontaneous Write Interface: When a database offers this interface for a data item X, it promises not to update X spontaneously. An event is called *spontaneous* if it can occur at any time, independent of constraint management. Spontaneous events model actions performed by local applications that may be unaware of the CM, and that operate on the local database independently. We use the event template $W_s(X, b)$ to represent an application performing the spontaneous write operation $X \leftarrow b$. The "no spontaneous writes" interface guarantees that there will be no $W_s(X, b)$ events. We express this in our interface specification language by using a special event $\mathcal{F}$ (for *false*), which, by definition, can never occur. Using $\mathcal{F}$, we write the following specification for this interface:

$$W_s(X, b) \rightarrow \mathcal{F}$$

Note that this interface does not mean that X can never be updated, only that it cannot be updated without involving the CM. (Data items may have more than one interface.)

Notify Interface: When a database offers a notify interface for a data item X, it promises to notify the CM within some time bound every time X is updated spontaneously. By using $N(X, b)$ to represent the CM receiving a notification of the update operation $X \leftarrow b$, and using δ to represent the time bound on notification, we express this interface as follows:

$$W_s(X, b) \rightarrow_\delta N(X, b)$$

Conditional Notify Interface: This is a refinement of the Notify Interface. In this interface, the database notifies the CM only when, in addition to X being updated spontaneously, some condition is satisfied. In addition to reducing communication costs, such an interface is useful when the local database can evaluate conditions that cannot be evaluated from the outside. A simple example is an interface that sends a notification to the CM only when the update changes the value of X by more than 10%. To express this interface, we use a spontaneous write event of the form $W_s(X, a, b)$, which represents X being updated from a to b. We then write the following:

$$W_s(X, a, b) \wedge (|b - a| > a * 0.1) \rightarrow_\delta N(X, b)$$

Periodic Notify Interface: Another kind of notify interface is one in which the current value of the data item is sent to the CM periodically. To describe such periodic interfaces, we use the notion of periodic events of the form $P(p)$, which occur every p seconds by definition. A 300-second periodic notify interface is expressed as follows:

$$P(300) \wedge (X = b) \rightarrow_\delta N(X, b)$$

This interface states that every time a $P(300)$ event occurs (every 300 seconds), a notification with the current value of X is sent to the CM within δ seconds.

Read Interface: When a database offers a read interface for a data item X, the CM can send it a read-request and the database will respond with the current value of X within some time bound δ. We use the event $RR(X)$ to represent the database receiving a read request from the CM, and the event $R(X, b)$ to represent the CM receiving the response from the database. We express this interface as follows:

$$RR(X) \wedge (X = b) \rightarrow_{\delta} R(X, b)$$

Parameterized Interfaces: In each of the above interfaces, the data item name X may be parameterized, yielding an interface for a set of related data items. For example, let $phone(n)$ denote "the phone number of n," where n is the name of an employee. Then, to specify that the CM is notified every time the phone number of any employee n is updated (spontaneously), we use a Parameterized Notify Interface, written as:

$$W_s(phone(n), b) \rightarrow_{\delta} N(phone(n), b)$$

3.2 Specifying Strategies

The strategy for a constraint describes the algorithm used by the constraint manager to monitor or enforce the constraint. Strategies are specified using a rule-based notation similar to that used for interfaces.

The strategy for a given constraint is defined by a set of *strategy statements* of the form:

$$\mathcal{E}_1 \rightarrow_{\delta} C?\mathcal{E}_2$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are event templates and C is a condition involving the events and data items local to the site of the event matching template $\mathcal{E}_2$. (Each event has a unique site.) This statement states that if an event matching template $\mathcal{E}_1$ occurs at time t, then there exists a time t' in the time interval $[t, t + \delta]$ such that if C is true at t' then an event matching template $\mathcal{E}_2$ occurs at time t'. The condition may be omitted when it is not needed. The events represented by templates $\mathcal{E}_1$ and $\mathcal{E}_2$ can be at different sites, however, the condition C can refer to data at the site of the right-hand side event only.

3.2.1 Example

Consider the copy constraint $X = Y$, where X and Y are at different sites. Let X have a Notify Interface (recall Section 3.1.1) and let Y have a Write Interface. A simple strategy in this case is the propagation of updates from X to Y by making a write request at Y whenever a notification is received from X. Assuming the write request can follow the notification within 5 seconds, we write:

$$N(X, v) \rightarrow_5 WR(Y, v)$$

Just like interfaces, strategies can be parameterized. For example, let $phone1(n)$ denote "the phone number of n," and let $phone2(n)$ denote the same phone number stored in another database.[2] We can specify that a write request is sent to $phone2(n)$ within 5 seconds every time a notification is received from $phone1(n)$ using the following rule:

$$N(phone1(n), v) \rightarrow_5 WR(phone2(n), v)$$

In general, our framework is capable of expressing more complex strategies than these examples. Each CM-Shell can have private data, stored in the CM-Shell itself, for use in strategies. This data may be read and written in the right-hand side of strategy rules. For example, the CM can use a local data item Cx to cache the value of X (obtained, for example, through a Periodic Notify Interface) using the following rule:

$$N(X, b) \rightarrow_5 W(Cx, b)$$

Note that Cx is a local data item maintained by the CM, while b is a parameter used only to express the rule. To forward a write request to a remote data item Y only when the new value of X differs from the cached value, we can write the rule:

$$N(X, b) \rightarrow_5 (Cx \neq b)?WR(Y, b)$$

The reader may observe that this and the previous rule are triggered by the same event, and that this rule must fire before the previous one. As explained in Appendix A.1, our rule language permits a sequence of conditions and events on the right-hand side of rules to achieve this. Note that the CM-Shell at each site can use only data that is local to that site, therefore strategies do not need global data access.

Once our framework has been used to specify a strategy (and to verify the correctness of a guarantee) then the rule-based strategy specification is implemented using the host language of the Constraint Manager. In our toolkit, the implementation uses a distributed rule engine, although other implementations could also be used.

3.3 Specifying Guarantees

A guarantee is essentially a modified (usually weakened) form of the constraint being managed. As we will see, guarantees vary in strength, from guarantees like "$X = Y$ always," which is very useful but very difficult to achieve, to guarantees like "$X = Y$ if there are no updates for a day," which is easy to achieve but not very useful. In between these two extremes is a spectrum of weakened guarantees that are both useful and relatively easy to achieve. One of the strengths of our approach is that it lets us specify guarantees anywhere in this range, unlike existing systems where one either

[2]Note that the two databases can be of different types. For example, the first may be a relational database while the second is a flat-file system. The complexity of translation to and from these different data models is handled by the *CM-Translators*, described in Section 4.

gets strong consistency (with distributed transactions, when applicable), or no consistency at all. However, we note that identifying the right weakened guarantees that are meaningful to applications and that can be enforced is challenging. We return to this issue in Section 7.1.

Guarantees are logical expressions involving occurrences of events and predicates over data items and time. The basic construct of the guarantee language is the following:

$$\{Event|Condition\}@Time\_variable$$

For example, $W(X,5)@t_1$ means that there is a write operation "$X \leftarrow 5$" performed at time t_1. Similarly $(X = 25)@t_2$ means that the data item X has value 25 at time t_2. In addition to this construct, we have predicates over data items, variables, and constants, and the usual logical connectives such as *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implies* ($\Rightarrow$), etc. We also permit a limited form of quantification for variables representing time, as described in the first example below. (Note that quantification over data involved in a constraint is achieved by means of parametrized data names, as explained in Section 3.1.1.)

3.3.1 Example: Some Guarantees for Copy Constraints

Consider the case of an inter-site copy constraint $X = Y$ between a data item X at site S_1 and a data item Y at site S_2. Suppose we wish to maintain Y as a copy of X. Below, we discuss some guarantees, one or more of which may be useful for a given application. For simplicity in presentation we consider a single copy constraint, but our guarantees also apply to a set of inter-site copy constraints over related data items, where X and Y are replaced by parameterized data names.

- A simple guarantee that is desirable in many situations is that at no time should Y have a value not previously taken by X. Informally, we call this the "Y follows X" guarantee. Formally, we express this by saying that if Y has a certain value at time t_1, then X must have had that value at some time t_2 before t_1. Note that, implicitly, variables on the left-hand side of the $\Rightarrow$ sign are universally quantified, while those on the right-hand side are existentially quantified:

  $(Y = y)@t_1 \Rightarrow (X = y)@t_2 \wedge (t_2 < t_1)$ ....(1)

- In some cases, one may also want that every value taken by X is eventually reflected in Y. That is, if $X = x$ at some time, we are guaranteed that $Y = x$ at some later time; there are no missing values. Informally, we call this the "X leads Y" guarantee. Formally, we express this as follows:

  $(X = x)@t_1 \Rightarrow (Y = x)@t_2 \wedge (t_2 > t_1)$ ....(2)

  Note that this guarantee may not be desirable in all situations. For example, if X represents the position of a "player" in an interactive distributed game, we usually are only interested in the latest position of the player (the most recent value of X), and we do not care about

a missed update. On the other hand, there are situations in which it is important for each value to be propagated. For example, if X represents the phone number of an employee, and if Y is a copy of X on another system that is interested in recording all the phone numbers of this employee over time, then guarantee (2) is desirable.

- In many cases, the order in which the updates are propagated is important in addition to the assurance that they are eventually propagated. For example, if X represents the position of a robot and Y is its copy on a system that plots the robot's path, we would like to receive the updated positions of the robot in the order in which the updates are actually made. Informally, we call this the "Y strictly follows X" guarantee. Formally, we express this as follows:

$$(Y = y_1)@t_1 \wedge (Y = y_2)@t_2 \wedge (t_1 < t_2) \Rightarrow (X = y_1)@t_3 \wedge (X = y_2)@t_4 \wedge (t_3 < t_4) \quad .....(3)$$

We call guarantees such as the ones above *non-metric* since they do not make explicit reference to time intervals. That is, they specify only the order in which events occur and predicates are satisfied, not an explicit delay between them. In contrast with non-metric guarantees, we also have *metric* guarantees, which state that some event must occur or some predicate must be true within some fixed time bound of another event or predicate. We can extend non-metric guarantees (1) and (2) above to metric guarantees by placing a bound on the delay between the time at which the two conditions mentioned in the guarantees are true. For example, the metric form of guarantee (1) is the following:

$$(Y = y)@t_1 \Rightarrow (X = y)@t_2 \wedge (t_1 - \kappa < t_2 < t_1) \quad .....(4)$$

where κ is a constant. This guarantee specifies that if $Y = y$ at time t_1, then $X = y$ at some time t_2 that is at most κ seconds before t_1. Informally, Y takes values held by X no more than κ seconds ago.

4 A Toolkit for Constraint Management

We have presented a framework and language for specifying interfaces, strategies, and guarantees for constraint management in heterogeneous systems. In this section we describe how we have used this framework to develop and implement a toolkit for constraint management. The toolkit provides a set of easily configurable services that monitor or enforce constraints spanning multiple loosely coupled databases. We first briefly describe the architecture of the toolkit, and then we use an example to illustrate some of its features.

4.1 Architecture

Figure 2 depicts the architecture of our constraint management toolkit, which is a realization of the logical architecture depicted in Figure 1. At the lowest level we have the *Raw Information Sources* (RIS), which could be relational or object-oriented database systems (OODBs), file systems, bibliographic information systems, electronic mail systems, network news systems, and so on. Each

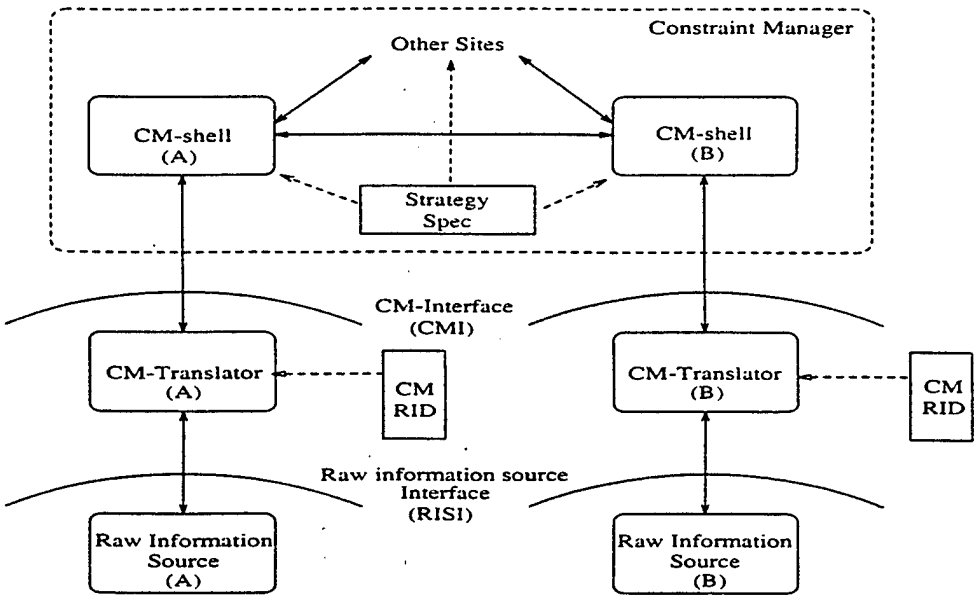


Figure 2: Constraint Management Toolkit Architecture

RIS has its own particular interface, which we call RISI. For example, for a Sybase RIS, the RISI is SQL-based and includes the protocols to send a query to the Sybase server and receive the results. The *CM-Shell* processes at the top of the figure implement the selected strategy, which is described in the Strategy Specification. Thus, each CM-Shell is a general-purpose process that is configured by reading the Strategy Specification file.

If the CM-Shell were to interact directly with the RIS, it would have to understand the peculiarities of each RISI. For example, to read a data item X stored in a relational database, a CM-Shell would have to issue a request in the particular dialect of SQL that the RIS understands. If X is stored in an OODB or a file system, the procedure to read X will be completely different. To factor this complexity away from the CM-Shells, we provide a CM-Translator (for each RIS) that presents to the CM-Shells the local capabilities in a standard fashion. This interface provided by the CM-Translator is the CM-Interface (CMI). The design and implementation of the CM-Translator is helped by the CM-Raw Interface Description (CM-RID) file, which configures standard CM-Translators to the particular underlying data source by presenting the specifics of the RISI in a standard format. For example, a CM-Translator for relational databases can be configured to interface with any DBMS (e.g., Sybase, Oracle) and any database (e.g., a payroll database, an inventory database) just by specifying the appropriate CM-RID.

A final component of our architecture (not shown in Figure 2) is a library of common interfaces and strategies. Thus, the contents of the Strategy Specification and the CM-RID files can usually be selected from available menus of proven strategies and interfaces. However, the toolkit is extensible and can accommodate custom interface and strategy descriptions written using our rule language.

During initialization, the CM-Shells query the CM-Translators about the local capabilities and

services. The CM-Translators respond with the interface specifications. The CM then suggests strategies that are applicable to these interfaces, along with the associated guarantees. The system administrator can either select one of the suggested strategies, or specify a different strategy using the strategy specification language. Once a strategy is specified, the CM distributes the rules of the strategy to CM-Shells based on the site of the event on the left-hand side of the rule. Each rule is executed in the CM-Shell handling the site at which the left-hand side event occurs. Based on this distribution of rules, the CM also determines, for each event template in each rule, the CM-Shells and/or the CM-Translators to which an event matching that template must be forwarded. During initialization, the CM-Translators also perform any set-up required for supporting the selected interface. For example, a CM-Translator supporting a Notify Interface for a Sybase RIS may need to declare triggers on the underlying database.

At run-time, the CM-Shells process events received from their respective CM-Translators and fire rules appropriately. The events that are produced as a result of rules firing are forwarded to the local CM-Translator and other CM-Shells as determined during initialization. CM-Translators implement the events using the native facilities of the RIS, thus executing the strategy. The CM-Shell supports a simple programmatic interface to allow applications to read auxiliary CM data for the guarantees that refer to it.

4.2 Example

Consider the following scenario. A company stores the personnel information for some of its employees in a local San Francisco branch database A. Personnel information also is stored in a database B at the headquarters in New York. These databases are loosely coupled in the sense described in Section 1. We wish to maintain the following constraint: For each employee in the San Francisco database, the salary stored in database A must equal the salary stored in database B. This is an example of a parameterized copy constraint. Let $salary1(n)$ denote the salary of n in database A, where, intuitively, n represents the employee ID. Similarly, let $salary2(n)$ denote the salary of n in database B. The constraint is then $salary1(n) = salary2(n)$ for all n in database A.

We first demonstrate how the toolkit is used to define interfaces for $salary1(n)$ and $salary2(n)$, and then we show how a simple strategy is specified and implemented. Finally, we discuss the validity of different guarantees, and we show how, with very little effort, we can continue to enforce the copy constraint even when the interface for $salary1(n)$ changes. The reader may wish to refer to Figure 2 as the description proceeds.

4.2.1 Interfaces

Suppose the RIS B is a Sybase relational database that provides a write interface for data item $salary2(n)$, defined in our language as $WR(salary2(n), b) \rightarrow_\delta W(salary2(n), b)$. In practice, this interface means that the RIS at site B can be instructed to write object $salary2(n)$. The CM-RID tailors the CM-Translator to handle such write requests. In addition to the interface statement, the CM-RID at B specifies the following:

- The command that has to be issued to the RIS to perform the write. In our example, the CM-RID specifies that to write a value b to *salary2*(n), the SQL query "UPDATE EMPLOYEES SET SALARY = b WHERE EMPID = n" must be sent to the SQL server. Note that we use the parameter n in the query. Our CM-Translator performs the necessary substitution given a particular instance of n.

- Low-level details of the protocol for querying the SQL server. In our example, the CM-RID indicates that the underlying RID is a Sybase database, and also specifies the network name of the Sybase server, the port number to connect to, the name of the machine on which it is running, etc. Using these details, the CM-Translator can send the SQL query to the RIS and receive the acknowledgment.

Suppose the RIS at site A offers a notify interface for data item *salary1*(n). This interface is defined by the rule $W_s(salary1(n), b) \rightarrow_\delta N(salary1(n), b)$. For the purpose of this example, let us assume that the notify interface is implemented by declaring a database trigger on the data items *salary1*(n). The CM-RID specifies what the CM-Translator at A needs to do to declare the trigger, and what it should expect to receive from the RIS when *salary1*(n) changes.

4.2.2 Strategy

Consider the following simple strategy: Make a write request to *salary2*(n) within δ seconds whenever a notification of a write to *salary1*(n) is received. We express this using our strategy specification language as follows:

$N(salary1(n), b) \rightarrow_\delta WR(salary2(n), b)$

This strategy specification is processed by both of the CM-Shells. The strategy specification also indicates where objects are located, i.e., that *salary1*(n) is at site A and that *salary2*(n) is at B. As explained earlier, from the site of the event template on the left-hand side of the rule, the toolkit can determine which CM-Shell is responsible for executing each rule. In our example, the CM-Shell at A is responsible for the left-hand side of the rule because *salary1*(n) is at that site. When the A CM-Shell receives a $N(salary1(n), b)$ event from its CM-Translator, it forwards the event to the B CM-Shell, since the B CM-Shell is responsible for the right-hand side of the rule. The B CM-Shell then sends the $WR(salary2(n), b)$ event to its local CM-Translator. Based on the expected maximum execution time of each CM-Shell and the maximum transmission time between CM-Shells, the database administrators can compute an estimate for δ, the time guarantee in the rule. (See the discussion on timing guarantees at the end of Section 5.)

4.2.3 Guarantees

Given the interfaces and the strategy above, we can prove that guarantees (1), (2) and (3) of Section 3.3.1 are all valid. We can also prove that the associated metric guarantee (4) is valid for an appropriate κ. Intuitively, it is easy to see why these guarantees are all valid; yet, there

are important details (such as a requirement for in-order message processing) that were discovered during the process of verification of the guarantee using our proof rules.

Now consider what happens if the administrator at site A decides to change the interface for data item *salary1*(n) from the above notify interface to a read interface (described in Section 3.1.1). Now the CM is no longer notified of updates to *salary1*(n); instead, the database at A only offers to respond with the current value of *salary1*(n) whenever it is requested. Since the only way to find out about changes to *salary1*(n) in this scenario is to periodically read the salaries, we must use a polling strategy. The simplest strategy is to periodically read *salary1*(n) and propagate the value read to *salary2*(n).[3] We express this strategy as follows:

$P(60) \rightarrow_\delta RR(X)$
$R(X,b) \rightarrow_\delta WR(Y,b)$

Recall that the event $P(60)$ represents a periodic event that occurs every 60 seconds.

Guarantees (1), (3) and (4) from Section 3.3.1 are valid in this scenario, while guarantee (2) is not. Intuitively, it is easy to see why guarantees (1), (3) and (4) are valid. The reason guarantee (2) is not valid is that since we are polling *salary1*(n) periodically, it is possible for us to "miss" updates when two or more updates to *salary1*(n) occur in the same polling interval.

4.3 Implementation Status

We have implemented CM-Translators for Unix files and relational databases. The translators are implemented using an object-oriented approach that requires only minor amounts of rewriting when moving to different kinds of raw sources (RIS). Currently, some low-level details for communicating with, say, Sybase must be embedded in the CM-Translator code. This code has to be rewritten to port the CM-Translator to, say, an Oracle database. However, the amount of code that needs to be rewritten is typically less than a page. Porting the CM-Translator to, say, a WAIS-like RIS involves incorporating the WAIS protocol for the submission of queries and the retrieval of results. We could avoid the need to rewrite the CM-Translator by enhancing the CM-RID format to include a scripting language such as Tcl [Ous90]; there is a tradeoff here between complexity in the CM-Translator and complexity in the CM-RID. The design and implementation of translators is in itself a difficult and interesting issue. While we currently are building translators by hand, we hope to soon exploit related work we are doing in the context of a query mediation project [PGMW95].

We have used our toolkit to implement several constraint management scenarios such as copy constraints for data with read, write and notify interfaces. Strategies include update propagation, polling, and the Demarcation Protocol (described in Section 6). We are currently implementing a large scenario for distributed constraints involving several databases at Stanford. The databases include the Stanford "whois" database, the Computer Science Department's custom personnel

[3] Note that we could certainly do better, for example by caching *salary1*(n) at the CM and propagating updates to *salary2*(n) only when the value of *salary1*(n) changes. In the interest of simplicity, we do not consider this strategy here.

database ("lookup"), the database group's Sybase database, and a bibliographic database. There are copy constraints for different personnel data such as phone numbers, addresses, etc., stored in the different databases. We also have referential integrity constraints, such as one that specifies that every paper authored by a Stanford database researcher as reported by the bibliographic database must also be mentioned in the Sybase database.

Using our toolkit, we coordinate the activities of the loosely coupled, heterogeneous databases without modifying the databases or the existing applications, thus maintaining database autonomy. Heterogeneity in the modes of database access and control is handled in a uniform way by describing interfaces using our rule language. Furthermore, incorporating new databases or changing the interface to an existing database requires very little work, since only the high-level interface and strategy specifications have to be modified (and can be chosen from a menu in most cases). Even though the databases in the system are not transactional, our toolkit provides a formal notion of data consistency that is useful in practice.

5 Failure Handling

In a distributed environment, especially a loosely coupled one, coping with failures is an important component of any coordinating software. We classify failures of the databases in our architecture into the following two types:[4] We say a database interface has had a *metric failure* when it is unable to honor the time bounds specified in the interface specifications. In such a scenario, the actions mandated by the interface statements are eventually performed, but not within the time bound specified. Such failures may be caused by the underlying database being overloaded or crashing. (In many cases, crashes can be mapped to metric failures if the database has some basic recovery facilities and can "remember" messages that need to be sent out upon recovery.) When a metric failure occurs on one or more of the sites involved in a constraint, the metric guarantees for that constraint are no longer valid. However, the non-metric guarantees continue to be valid, which may allow many applications to continue to function.

The second kind of failure is one in which the interface statements are no longer valid at all. We call this a *logical failure*. Such a failure may be caused by catastrophic failure of one or more of the databases, and we expect such failures to be very infrequent. When a logical failure occurs, both metric and non-metric guarantees involving the failed site are no longer valid until the system is reset.

In our current implementation, failures are detected and flagged. In the future, we plan to incorporate a more sophisticated failure handling scheme into our toolkit, permitting applications to deal with failures in a more sophisticated manner. Recall that in our toolkit, the CM-Translator translates the raw interface (RISI) of the underlying database to the CM-Interface presented to the CM-Shell. The CM-Translator also maps (when possible) failures of the RISI into metric or logical

[4]Throughout this paper, we assume a reliable network. Therefore, we consider only site (database) failures here. Of course, network failures can be viewed as the failure of the sites sending the affected message.

failures of the CM-Interface. On detecting a failure, the CM-Translator notifies the local CM-Shell, which then propagates the information to other CM-Shells so that the affected guarantees may be marked as invalid.

The method used by the CM-Translator to detect failures of the RISI depends on the nature of the RISI and the CM-Interface being supported. For example, consider a CM-Translator implementing a Read Interface with a Unix file system as the underlying database. In this case, the the CM-Translator will use the *read()* system call interface to the Unix file system. Failure of this system call can be detected based on the return value, and such a failure can be flagged as a failure of the Read Interface. Depending on the reason for the failure of the *read()* call, the Read Interface failure is flagged as either a metric or a logical failure. Similarly, a CM-Translator for a Sybase database can detect and flag interface failures based on the error codes returned by calls to the Sybase library routines that communicate with the SQL server.

Sometimes, however, the nature of the RISI may make detecting failures very difficult or impossible. For example, consider a CM-Translator supporting a Notify Interface for a legacy database, and suppose the database simply sends a message to the CM-Translator whenever there is an update to some data item. If the database fails silently and does not report some update, there is no way for the CM-Translator to detect the failure. If it is not possible to ensure that the probability of such undetectable failures is acceptably low, then a Notify Interface should not be used for this database. Often, one can use another available interface, such as a Read Interface, and use polling to simulate notification, as in the example in Section 4. Note that some probability of undetectable failures exists in most systems. For example, undetected hardware failures (e.g., double parity errors on disk or memory reads) can silently corrupt data on disk or in memory, and this is not detected until some application fails unexpectedly.

The probability of a metric failure of an interface depends on the choice of the time bound in the rules specifying that interface. Typically, these time bounds would be determined based on the processing power of the information source, the expected load, the maximum estimated communication delay (including retries), etc. In practice, these constants could be chosen to ensure that the interface is honored with, say, 99.99% probability. It is well known that all fault tolerant systems have to select constants (timeouts, number of retries, etc.) in a similar manner [Cri89]; the difference is that our toolkit makes the effects of choosing these constants explicit in the form of metric guarantees, so that applications do not have to guess.

6 Additional Constraint Management Scenarios

Our framework and toolkit for constraint management can be applied in a variety of scenarios, with interfaces and strategies ranging from simple (such as those described so far) to complex. In this section, we briefly illustrate some additional scenarios to show the wide range of interfaces, strategies and guarantees that our framework and toolkit can cover.

6.1 Demarcation Protocol

Consider the inequality constraint $X \leq Y$ where X and Y are at different sites. The *Demarcation Protocol* [BGM92] is applicable in this scenario if the sites of X and Y both offer certain interfaces. The protocol guarantees that the constraint $X \leq Y$ is always valid. The protocol uses local *limit data items* X_l and Y_l (located at the sites of X and Y respectively) and uses the constraint managers of the underlying databases to enforce the local constraints $X \leq X_l$ and $Y \leq Y_l$. Using our framework, we can accurately specify the interfaces assumed by the Demarcation Protocol, which are fairly complex. We can also specify the Demarcation Protocol itself. Then, using our proof rules, we can prove the guarantee $X \leq Y$ based on the specification of the protocol strategy and the interfaces. Thus, our framework is capable of expressing a complex scenario, in which we use the facilities of the underlying database (such as local constraint managers) in order to implement a global constraint, in addition to the simple ones we discuss elsewhere in this paper.

There are many ways of implementing the Demarcation Protocol such that the above guarantee is valid, and some of these are less desirable than the others. For example, an implementation that simply does not change the limit data items X_l and Y_l will satisfy the above guarantee, but is not very desirable since it does not permit X (and Y) to ever exceed (respectively, fall below) their original limit values. We can formalize this intuitive guarantee by introducing an event to denote a request for a limit-change operation, and by specifying that if there is enough "slack" at one site, then a change-limit request at the other site must be granted within some time. Different implementations of this protocol (called *policies* in [BGM92]) can then be compared using this guarantee. In [CGMW94], we have presented a formal specification of interfaces, strategies and guarantees for the Demarcation Protocol.

6.2 Referential Integrity

Consider a referential integrity constraint that states that for every employee ID having a record describing a project assignment ("project record" for short) in one database, there must be a record in another database with the salary information for that employee ID ("salary record" for short). A weakened form of this constraint, more suitable to loosely coupled heterogeneous systems, is the following guarantee: The above referential integrity constraint may be violated for any one employee ID for a period of at most 24 hours. We express this guarantee using an *exists* predicate.[5] An exists predicate $E(X)$, where X is a data item name (usually parameterized), is true if and only if the data item X exists in the database. In our example, let *project*(i) denote the project record of an employee with employee ID i, and let *salary*(i) denote the salary record of that employee as stored in the second database. The guarantee states that if the project record exists at time t in the first database, the salary record must exist in the second database within 24 hours (86,400

[5] We use an explicit exists predicate because our language does not include general quantification for data item names (although we do have quantification of time variables).

seconds), and is expressed as follows:

$E(project(i))@t \Rightarrow E(salary(i))@(t + 86400)$

A simple strategy to realize the above guarantee is the following. At the end of each working day, the CM deletes all project records from the projects database that do not have a corresponding salary record in the salary database, perhaps notifying the database owner of the deleted records. This strategy assumes the projects database permits the CM to delete records. If this is not the case, then there may be no way for the CM to enforce the referential integrity constraint. However, the CM could still monitor the constraint using a technique similar to that described in the next section.

6.3 Monitor

Consider a scenario where we have a copy constraint $X = Y$, both X and Y have notify interfaces (see Section 3.1.1), and the CM cannot update either data item. In this case, the best the CM can do is to monitor the constraint. One method of monitoring the constraint is to maintain some auxiliary data items at the site of the application[6] interested in this constraint. (We discuss the storage and access of auxiliary data in Section 7.1.) These auxiliary data items indicate the validity of the constraint over time. In particular, we may offer the following guarantee to an application:

$((Flag = true) \wedge (Tb = s))@t \Rightarrow (X = Y)@@[s, t - \kappa]$

Here *Flag* and *Tb* are auxiliary data items The guarantee states that if *Flag* is true, then $X = Y$ was true at all times in the time interval indicated on the right-hand side. The auxiliary data item *Tb* is used to keep track of the start of the time interval during which $X = Y$. (In the guarantee, we use *Tb* by accessing its value s on the right-hand side.) Details of the strategy used to ensure this guarantee, as well as the proof of the guarantee using the interface and strategy specifications is in [CGMW94].

6.4 Periodic Guarantees

A periodic guarantee is one that states that the constraint is valid periodically. For example, consider an old-fashioned banking environment in which all update transactions occur between 9 a.m. and 5 p.m. Suppose we have a set of copy constraints stating that balances for each account at the local branch and the head office must be equal. A simple strategy in this situation is to propagate the new values of account balances from the branch to the head office at the end of each working day. If the branch offers an interface that guarantees that there will be no updates to account balances between 5 p.m. and 8 a.m., and if the propagation of new values at the end of the day takes 15 minutes, we can offer a periodic guarantee that the copy constraints will be valid every day from 5:15 p.m. to 8 a.m. the next day. Such a guarantee permits, for example, a financial

[6] In the case of multi-site applications, this is the site of the CM-Shell that services the application. The choice of a CM-Shell can be arbitrary.

analysis application at the main office to proceed with the assurance of consistency, assuming it runs in the above time interval.

7 Discussion

In this section, we discuss some details of how applications can use the guarantees offered by our toolkit. We also explain why the implementation of strategies does not require global data access and clocks, and we discuss how auxiliary CM data is stored and accessed.

7.1 Using Guarantees

From the viewpoint of an application, weakened notions of consistency as expressed by our guarantees are not as easy to use as conventional, strict consistency is. Yet, given the restrictions on data access and control in a loosely coupled heterogeneous environment, weakened consistency is usually all that can be offered. Weakened consistency is certainly more useful than no consistency guarantees at all, which is what usually is offered to applications in current systems. In this section, we discuss how applications can use weakened consistency guarantees.

The ease of use of a guarantee depends on the "strength" of the guarantee. Guarantees that are relatively strong are easy to use. For example, a strong non-metric guarantee like $X \leq Y$ (as offered by the Demarcation Protocol) permits the application know at all times that the data it sees is consistent. Similarly, consider guarantees (1) and (2) of Section 3.3.1 from the viewpoint of an application that runs at Y's site and tabulates the different values taken by X. This application can read Y and be assured that Y is a value previously taken by X (due to guarantee (1)) and that Y does not miss any values that X takes (due to guarantee (2)). Hence, these guarantees permit the application to know that its tabulation is correct.

The guarantees that are harder to use are those that are conditional on the values of some auxiliary data items. For example, consider the following guarantee introduced in Section 6.3:

$$((Flag = true) \wedge (Tb = s))@t \Rightarrow (X = Y)@@[s, t - \kappa]$$

In order to use this guarantee, the application must read the values of the auxiliary data items *Flag* and *Tb*. Such auxiliary data may be stored in a private database of the CM-Shell at the site of the application. In this case, the application reads their values through the CM-Shell. Alternatively, the underlying database at the site may offer to store such auxiliary data for the CM-Shell. In this case, the application can read the auxiliary data directly from the database. In our toolkit, for example, auxiliary data for a CM-Shell that is at the site of a Sybase database is stored in the Sybase database, while the auxiliary data for a CM-Shell that is at the site of a bibliographic information system is stored in a private database.

Note that guarantees are always designed in such a way that that applications can use them without the need for global transactions. This is done by ensuring that reading local data only is sufficient to conclude some desirable properties of the global data. In the above guarantee, for

example, by reading local (auxiliary) data items *Flag* and *Tb*, an application can determine the validity of the global constraint $X = Y$. It is a straightforward extension of the strategy and guarantee in that example to replicate *Flag* and *Tb* at the site of each application interested in the guarantee, so that each application accesses only local data.

The reader will note that it is important to read the data items on the left-hand side of the $\Rightarrow$ in the guarantee consistently. If the auxiliarly data items *Flag* and *Tb* are stored in the CM-Shell, the CM-Shell ensures that they are read consistently, since they are under its control. If, however, they are stored in the underlying database, then the database must have some facility to permit consistent read of these data items.

Once the application reads the auxiliary data items *Flag* and *Tb*, it can determine whether the constraint was valid during the interval specified on the right-hand side of the guarantee. To see how this guarantee is useful, suppose that the application received some query results based on the values of X and Y at some time in the past. Then this guarantee permits the application to check whether that the query was computed based on a consistent state of the data. If $X = Y$ was true at the time at which the earlier query was computed, the application can proceed with confidence in the query results. If the guarantee is inconclusive about whether $X = Y$ was true at the query-computation time (either because *Flag* is false or because the time interval on the right-hand side of the guarantee does not include the time of interest to the application), the application can either proceed with the understanding that the query results may not be accurate, or it can recompute the query in the hope that the new computation will be performed on a consistent state.

In this paper, we have illustrated guarantees only for some simple copy, inequality, and referential integrity constraints. As constraints get more complex, their guarantees will also increase in complexity, making them more difficult to use. However, note that simple constraints like the ones we consider here are the most common kind of constraints in a loosely coupled heterogeneous environment, where it is unlikely that autonomous data repositories will have very complex interdependencies. Furthermore, if there are complex constraints in a loosely coupled heterogeneous system, they are often split into distributed copy constraints plus local constraints. For example, consider the constraint $X = Y + Z$, where X, Y, and Z are at three different sites. A common way to manage this constraint is to have cached copies Y_c and Z_c of Y and Z, respectively, at the site where X is. Hence, we would have the constraints $X = Y_c + Z_c$, $Y_c = Y$ and $Z_c = Z$. Only the simple copy constraints are distributed and they can be handled by the strategies of Section 3.3.1, for instance. Thus, even if our framework is used for simple constraints only, we believe it can cover the vast majority of the scenarios of interest for loosely coupled heterogeneous systems.

7.2 Executing Strategies

In Section 7.1 we have seen how the use of guarantees does not require global transactions. Similarly, execution of strategies (rules) does not require global transactions. To see this, note that while the left-hand side and right-hand side of a rule each (separately) execute "atomically," the entire rule does not. Further, each side of a rule is restricted to accessing data that is all at the same site.

Thus the atomic execution of each side of the rule can be implemented in the local CM-Shell.

Another issue is that of clock synchronization. In the formal part of our work, we use global time to reason about events and conditions. This approach is similar to that in [Koy92]. For example, consider a rule of the form $\mathcal{E}_1 \rightarrow_5 \mathcal{E}_2$ where events E_1 and E_2 matching the event templates $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, are at different sites. This rule states that if E_1 occurs at site S_1 at 9:00:00 a.m., then E_2 must occur at site S_2 before 9:00:05 a.m., where both times refer to absolute, wall-clock time. Recall that this is a specification of an interface or strategy based on expected maximum delays. Implementing such a rule does not require any access to global time (or even a notion of time in the implementation). Thus our toolkit does not rely on global clock synchronization for implementing strategies even though we use it as a reasoning tool in our formal framework. Certain kinds of guarantees, such as the periodic guarantees of Section 6, explicitly refer to global time, and they assume global clocks. Such a scenario does not pose a problem as long as the time intervals specified in the guarantee are significantly larger than the expected skew in system clocks. In the example of Section 6, a clock skew of a few seconds (or even minutes) can be accommodated by including an error margin in the interval specified in the guarantee.

8 Conclusion

Distributed integrity constraints arise naturally when information systems inter-operate, due to interdependencies between data. Traditional constraint management techniques assume facilities like atomic transactions, locking, and global queries. While these are reasonable assumptions in centralized or tightly coupled distributed environments, they typically do not hold in loosely coupled heterogeneous environments, and traditional constraint management techniques are therefore inapplicable in such cases. Another characteristic of heterogeneous environments is that different databases offer different facilities and capabilites for accessing data, which also makes constraint management more difficult. Currently, constraints in heterogeneous environments are either not monitored at all, or are monitored using ad-hoc techniques. Such techniques are error-prone and can lead to irreparable inconsistencies in the databases.

We have presented a framework and a toolkit for constraint management in loosely coupled, heterogeneous information systems. Our framework formalizes weakened notions of consistency, which are essential in real-world loosely coupled heterogeneous scenarios, where it is not possible to guarantee strict consistency. Our framework also allows us to formally specify the interfaces each database offers, along with constraint management strategies. Our toolkit provides a set of configurable modules that enable us to monitor and enforce constraints in a uniform and useful manner.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGM92] D. Barbara and H. Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 373–388, Vienna, Austria, March 1992.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181, October 1992.

[CGMW93] S. Chawathe, H. Garcia-Molina, and J. Widom. Constraint management in loosely coupled distributed databases. Technical report, Computer Science Department, Stanford University, 1993. Available through anonymous ftp from host `db.stanford.edu` as `pub/chawathe/1993/cm-loosely-coupled-dbs.ps`.

[CGMW94] S. Chawathe, H. Garcia-Molina, and J. Widom. Constraint management in loosely coupled distributed databases. Technical report, Computer Science Department, Stanford University, 1994. Available through anonymous ftp from host `db.stanford.edu`.

[Cri89] F. Cristian. A probabilistic approach to distributed clock synchornization. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 288–296, Jun 1989.

[CW93] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the International Conference on Very Large Data Bases*, pages 108–119, Dublin, Ireland, August 1993.

[Elm91] A. Elmagarmid, editor. *Special Issue on Unconventional Transaction Management*, Data Engineering Bulletin 14(1), March 1991.

[Gre93] P. Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In *Proceedings of the International Conference on Very Large Data Bases*, pages 581–591, Dublin, Ireland, August 1993.

[GW93] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, Washington, D.C., May 1993.

[GW94] P. Grefen and J. Widom. Integrity constraint checking in federated databases. Memoranda Informatica 94-80, Department of Computer Science, University of Twente, The Netherlands, December 1994.

[Koy92] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Number 651 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[L+93] D. Luckham et al. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

[Ous90] J. Ousterhout. Tcl: an embeddable command language. In *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 1990.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources, March 1995.

[RSK91] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, 24(12):46–51, December 1991.

[SV86] E. Simon and P. Valduriez. Integrity control in distributed database systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 621–632, 1986.

Appendix A Syntax and Semantics of Rule Language

In Section 3, we motivated and informally presented the rule language used in our framework. Many concepts were introduced by example in that section. In this section, we present the formal specification of the syntax and semantics of our rule language. Examples of how these definitions can be used to prove guarantees of consistency can be found in [CGMW94].

Appendix A.1 Events, Templates and Rules

We first define events and event templates, which are the building blocks of our rule langage. The syntax and informal semantics of the rule language are presented next. (Formal semantics of the rule language are in the next section.)

Let $\{D_1, D_2, \ldots, D_n\}$ be the set of all data items in the system, including all the databases and any data stored by the constraint manager. An *interpretation* I is a function that maps each D_i to a value, yielding a state of the system. For example, if we have three data items $\{D_1, D_2, D_3\}$, then a possible interpretation is $\{D_1 = 7, D_2 = 14, D_3 = 49\}$. We permit an interpretation to "under-specify" the state by allowing some data items to map to null, meaning these data items can assume any value. The system passes through a sequence of states, each represented by an interpretation of its data items.

The behavior of the databases and the constraint manager is described by *events*. For the purposes of constraint management, we divide events into two types:

- *Spontaneous events*, which occur as a result of users or application programs operating on the databases.
- *Generated events*, which occur as a (direct or indirect) result of a strategy being executed by the CM or an interface being maintained by a database.

Each event is represented using a six-tuple: E = (time, desc, old, new, rule, trigger), where the components of the tuple are described below:

time: The time at which the event E occurs. For simplicity, we assume that all references are to global "physical" time. We use time mainly for reasoning about correctness, and as we will see, in practice we do not require synchronized clocks.

desc: The descriptor of the event, drawn from the following set of descriptors. (This set can be expanded by adding new templates and their semantics.)

$$\{W_s(x,\_\_), W_g(x,\_\_), RR(x), N(x,\_\_), WR(x,\_\_)\}$$

old: The interpretation representing the state of the system just before the event occurs.

new: The interpretation representing the state of the system just after the event occurs.

rule: If E is a generated event, this is a rule whose "firing" resulted in the occurrence of this event. If E is a spontaneous event, this component is null. Rules are described below.

trigger: If E is a generated event, this is the event which caused the rule above to fire. If E is a spontaneous event, this is null.

For an event E, we denote a component of the event using dot notation. For example, $E.old$ denotes the *old* component of the event E.

We define an *event template* to be an event descriptor in which some of the components are parameterized or "wild-carded." An event template represents the set of all event descriptors that can be obtained from the template by substituting particular values for the parameters and wild-cards. For example, $W_s(X, b)$ represents the infinite set of spontaneous write event descriptors that have X as the first component and any value as the second component. We use "_" to denote a wild-card—a parameter whose name is not important. Thus $W_g(\_,\_)$ represents the set of all generated write event descriptors (of any value to any data item in the system). We use $\mathcal{E}$ to denote event templates. In the sequel we use $W_s(X, b)$ as shorthand for $W_s(X, \_, b)$.

We now define what it means for an event to *match* an event template. We say an event E matches an event template $\mathcal{E}$ if there is an *interpretation* I of the variables in $\mathcal{E}$ such that substituting using I in $\mathcal{E}$ yields E. Such a matching interpretation I, if it exists, is denoted by $mi(E, \mathcal{E})$. The special *false* event template, $\mathcal{F}$, does not match any event (by definition).

The general form of a rule is

$$\mathcal{E}_0 \wedge C_0 \rightarrow_\delta C_1?\mathcal{E}_1, C_2?\mathcal{E}_2, \ldots, C_k?\mathcal{E}_k$$

where $\mathcal{E}_i$ are event templates, C_0 is a boolean expression involving data items local to the site of $\mathcal{E}_1$ and variables, and C_i are boolean expressions involving data items local to the site of $\mathcal{E}_1$ and variables.[7] The meaning of this rule is as follows: If an event matching the event template on the LHS occurs at a time t at which the condition C_0 is true, then there exist $t_i \in [t, t+\delta]$, $\quad i = 1 \ldots k$ where $t_i < t_{i+1}$, $\quad i = 1 \ldots k-1$ such that at time t_i, the condition C_i is evaluated, and if it evaluates to true, the event matching event template $\mathcal{E}_i$ occurs. The event corresponding to the event template $\mathcal{E}_i$ is obtained by substituting in $\mathcal{E}_i$ using the matching interpretation for the LHS, $mi(E_0, \mathcal{E}_0)$. Note that variables on the LHS are implicitly universally quantified, while variables on the RHS that do not occur on the LHS are implicitly existentially quantified. The bindings of variables from the LHS are passed on to the RHS (through the matching interpretation), so that a variable occurring on both sides takes on the same value.

Appendix A.2 Valid Executions

We define the semantics of our rule language using the concept of valid executions over a system of databases. A *valid execution* is an execution $(E_1, \ldots, E_n)$ that satisfies the following properties. Note that these properties reflect the semantics of rules described in Section 3 and Appendix A.1.

[7] All the events on the RHS of a rule must have the same site.

1. The events in the sequence are sorted in order of nondecreasing time.

   $\forall i, j \in [1, n], \quad E_i.time < E_j.time \Rightarrow i < j$

2. For each event in the execution: If the event descriptor is a (spontaneous or generated) write, then the *new* interpretation maps the corresponding data item to the value written, with all other data items being mapped to the same value in both *old* and *new*. If the event descriptor is not a write, then the *old* and *new* interpretations are identical. Formally,

   $\forall i = 1 \ldots n,$
   if $E_i.desc = W_s(X, \alpha, \beta)$ then $E_i.new = E_i.old - \{X = \alpha\} \bigcup \{X = \beta\}$
   else if $E_i.desc = W_g(X, \beta)$ then $E_i.new = E_i.old - \{X = \_\} \bigcup \{X = \beta\}$
   else $E_i.new = E_i.old.$

3. The *old* interpretation in each event is identical to the *new* interpretation in the immediately preceding event. That is, the only changes to the interpretations are those caused by events. Note that interpretations model only data items related to constraints, hence this restriction applies to only such constraint data; other data items may change their values.

   $E_i.old = E_{i-1}.new \quad i = 2 \ldots n$

4. For all $i = 1 \ldots n$, if E_i is a spontaneous event then both $E_i.rule$ and $E_i.trigger$ are null.

5. For all $i = 1 \ldots n$, if E_i is a generated event, then (informally) its rule component specifies the rule whose firing caused E_i to occur, and its trigger component specifies the event whose occurrence caused the rule to fire. Further, the LHS and RHS conditions of the rule must be satisfied by the appropriate interpretations.

   Formally, if E_i is a generated event then both $E_i.rule$ and $E_i.trigger$ are non-null. Further, the following properties are true:

   (a) $E_i.trigger$ is an event that matches the LHS event template of $E_i.rule$. Let the matching interpretation be I;

   (b) I can be extended[8] to an interpretation I' such that substituting using I' in a RHS event template $\mathcal{E}_j$ of $E_i.rule$ gives E_i;

   (c) The LHS condition of $E_i.rule$ is satisfied by $E_i.trigger.new$;

   (d) The RHS condition C_j (corresponding to $\mathcal{E}_j$) of $E_i.rule$ is satisfied by $E_i.old$.

6. Informally, the converse of the previous property. That is, if an event matching the LHS event template of some rule occurs and if the LHS and (some) RHS conditions of that rule are satisfied at the appropriate times, then events matching the corresponding RHS event templates occur within the time specified by the rule.

[8] We say an interpretation I is *extended* to an interpretation I' if the set of non-null mappings in I is a subset of the set of non-null mappings in I'.

Formally, if E_i matches the LHS of a rule r: $\mathcal{E}_0 \wedge C_0 \rightarrow C_1?\mathcal{E}_1 \ldots C_k?\mathcal{E}_k$; $B \leq \delta$, and $E_i.new$ satisfies C_0, then there exist $t_j \in [E_i.time, E_i.time + \delta], j = 1 \ldots k$ where $t_j < t_{j+1}, j = 1 \ldots k-1$ and, for all $j = 1 \ldots k$, exactly one of the following holds true:

- C_j is false at t_j;
- there exists an event E_j, such that $E_j.time = t_j$, $E_j.old$ satisfies C_j, and substituting using matching interpretation $mi(E_i, \mathcal{E}_0)$ in $\mathcal{E}_j$ gives $E_j.desc$. Further, $E_j.rule = r$ and $E_j.trigger = E_i$.

7. This property formalizes our assumptions of in-order message delivery between sites and in-order processing at each site. To do this, we first introduce some additional notation.

   If E_i and E_j are events in an execution such that $E_j.trigger = E_i$ and $E_j.rule = R$ we write $E_i \rightsquigarrow^R E_j$.

   We say rules R_1: $\mathcal{E}_1^1 \wedge C_1 \rightarrow C_2?\mathcal{E}_2^1$ and R_2: $\mathcal{E}_1^2 \wedge C_1 \rightarrow C_2?\mathcal{E}_2^2$ are *related* if $site(\mathcal{E}_1^1) = site(\mathcal{E}_1^2)$ and $site(\mathcal{E}_2^1) = site(\mathcal{E}_2^2)$.

   The formal statement of this property is that if $E_1@t_1 \rightsquigarrow^{R_1} E_2@t_2$ and $E_3@t_3 \rightsquigarrow^{R_2} E_4@t_4$, where R_1 are R_2 are related rules, then $t_1 < t_3$ iff $t_2 < t_4$.

Using the above specification of the semantics of our rule language, in [CGMW94] we derive proof rules and present proofs of some consistency guarantees.

TSIMMIS Publications

All TSIMMIS papers and talks are available in postscript. If you need a viewer, check out Ghostview --- it's free and runs on Unix, Macintosh, DOS, and Windows.

Overview

- J. Hammer, H. Garcia-Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. "Information Translation, Mediation, and Mosaic-Based Browsing in the TSIMMIS System". In *Exhibits Program of the Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 483, San Jose, California, June 1995.

- H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and Jennifer Widom. "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS". In *Proceedings of the AAAI Symposium on Information Gathering*, pp. 61-64, Stanford, California, March 1995.

- Y. Papakonstantinou, H. Garcia-Molina and J. Widom. "Object Exchange Across Heterogeneous Information Sources". *IEEE International Conference on Data Engineering*, pp. 251-260, Taipei, Taiwan, March 1995.

- S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. "The TSIMMIS Project: Integration of Heterogeneous Information Sources". In *Proceedings of IPSJ Conference*, pp. 7-18, Tokyo, Japan, October 1994.

- S. Chawathe, H. Garcia-Molina and J. Widom. "Flexible Constraint Management for Autonomous Distributed Databases". *IEEE Data Engineering Bulletin*, Vol. 17, No. 2, pp. 23-27, June 1994.

Semistructured Data

- D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. "Querying Semistructured Heterogeneous Information". In *International Conference on Deductive and Object-Oriented Databases*, 1995.

Wrappers

- J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, R. Yerneni. "Template-Based Wrappers in the TSIMMIS System". In *Proceedings of the Twenty-Sixth SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 12-15, 1997.

- Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, J. Ullman. "A Query Translation Scheme for Rapid Implementation of Wrappers". In *International Conference on Deductive and Object-Oriented Databases*, 1995.

- J. Hammer, H. Garcia-Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. "Information Translation, Mediation, and Mosaic-Based Browsing in the TSIMMIS System". In *Exhibits Program of the Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 483, San Jose, California, June 1995.

- A. Rajaraman, Y. Sagiv, and J. Ullman. "Answering Queries Using Templates with Binding Patterns". In *Proceedings of the 14th ACM PODS*, pp. 105-112, San Jose, California, May 1995.

Web Extraction

- J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. "Extracting Semistructured Information from the Web". In *Proceedings of the Workshop on Management of Semistructured Data*. Tucson, Arizona, May 1997.
(Needs Ghostview 2.0 or higher.)

Mediators

- Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey Ullman, Murty Valiveti. "Capability Based Mediation in TSIMMIS". SIGMOD 98 Demo, Seattle, June 1998.

- V. Vassalos , Y. Papakonstantinou. "Describing and Using Query Capabilities of Heterogeneous Sources". In *VLDB Conference,* Athens, Greece, August 1997.

- H. Garcia-Molina , Y. Papakonstantinou , D. Quass , A. Rajaraman , Y. Sagiv , J. Ullman , V. Vassalos , J. Widom. "The TSIMMIS approach to mediation: Data models and Languages". In Journal of Intelligent Information Systems, 1997.

- S. Abiteboul , H. Garcia-Molina , Y. Papakonstantinou , R. Yerneni. "Fusion Query Optimization". *Technical Report*, 1996.

- Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina. "Object Fusion in Mediator Systems". In *VLDB Conference,* Bombay, India, September 1996.

- Y. Papakonstantinou, H. Garcia-Molina, J. Ullman. "Medmaker: A Mediation System Based on Declarative Specifications". In *International Conference on Data Engineering*, pages 132 - 141, New Orleans, February, 1996.

Information Browsing

- J. Hammer, R. Aranha, K. Ireland. "Browsing Object-Based Databases Through the Web". Technical report, Stanford University, October 1996.
(Needs Ghostview 2.0 or higher.)

Constraint Management

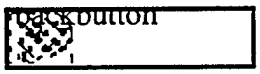
- S. Chawathe, H. Garcia-Molina and J. Widom. "A Toolkit for Constraint Management in

Heterogeneous Information Systems". In *IEEE Twelfth International Data Engineering Conference*, pp. 56-65, New Orleans, February 1996.

- S. Chawathe, H. Garcia-Molina and J. Widom. "Flexible Constraint Management for Autonomous Distributed Databases". *IEEE Data Engineering Bulletin*, Vol. 17, No. 2, pp. 23-27, June 1994.

Architecture & Implementation

- H. Garcia-Molina and A. Paepcke. "Proposal for I**3 Client Server Protocol". *Technical Report*, September 1996.



home page.